THE UNIVERSITY
of EDINBURGH

Approximating Neural Machine Translation for Efficiency

Alham Fikri Aji

Doctor of Philosophy
Institute for Language, Cognition and Computation
School of Informatics
University of Edinburgh
2020

Abstract

Neural machine translation (NMT) has been shown to outperform statistical machine translation. However, NMT models typically require a large number of parameters and are expensive to train and deploy. Moreover, its large model size makes parallel training inefficient due to costly network communication. Likewise, distributing and locally running the model for a client-based NMT model such as a web browser or mobile device remains challenging. This thesis investigates ways to approximately train an NMT system by compressing either the gradients or the parameters for faster communication or reduced memory consumption. We propose a gradient compression technique that exchanges only the top 1% of the most significant gradient values while delaying the rest to be considered for the next iteration. This method reduces the network communication cost by 50-fold but causes noisy gradient updates. We also find that Transformer–the current state-of-the-art NMT architecture–is highly sensitive to noisy gradients. Therefore, we extend the compression technique by restoring the compressed gradient with locally-computed gradients. We obtained a linear scale-up in parallel training without sacrificing model performance. We also explore transfer learning as a better method of initialising the training. With transfer learning, the model converges faster and can be trained with more aggressive hyperparameters. Lastly, we propose a log-based quantisation method to compress the model size. Models are quantised to 4-bit precision with no noticeable quality degradation after re-training combined with reserving the quantisation errors as feedback.

Lay Summary

Machine translation is an automatic process of translating a text from one language to another (i.e., English to German). Over the years, more advanced models have been developed, which improve quality but also more resource-intensive. Training a machine translation model with multiple computers is inefficient since each computer must communicate hundreds of megabytes across the network for each training step. Notably, a machine translation model can take more than 100,000 steps to train. Likewise, using the system for offline use (such as for web-based or mobile devices) is impractical since users must download the model, which can be hundreds of megabytes in size. This thesis focuses on exploring methods of efficiently training and deploying machine translation through approximation. We can significantly cut network communication costs in parallel training by only exchanging 1% of the most significant information, making the communication 50x more efficient. Instead of training the model from scratch, we also explore transfer learning, in which an already trained model can be used and adjusted with the new language pairs to hasten the training process. Lastly, we also explore methods to train a model with lower mathematical precision so it can be stored and distributed with reduced memory size.

Acknowledgements

This thesis is funded by the Indonesia Endowment Fund for Education scholarship scheme.

The work in Chapter 3, 5, 6 and 7 were performed using resources provided by the Cambridge Service for Data Driven Discovery (CSD3) operated by the University of Cambridge Research Computing Service (http://www.csd3.cam.ac.uk/), provided by Dell EMC and Intel using Tier-2 funding from the Engineering and Physical Sciences Research Council (capital grant EP/P020259/1), and DiRAC funding from the Science and Technology Facilities Council (www.dirac.ac.uk). Computing for work in Chapter 4 was funded by the Amazon Academic Research Awards program and by Microsoft's donation of Azure time to the Alan Turing Institute.

The research in Chapter 6 is based upon work supported by the Office of the Director of National Intelligence (ODNI), Intelligence Advanced Research Projects Activity (IARPA), via contract # FA8650-17-C-9117. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of ODNI, IARPA, or the U.S. Government. The U.S. Government is authorised to reproduce and distribute reprints for governmental purposes notwithstanding any copyright annotation therein.

The work in Chapter 7 was conducted within the scope of the Horizon 2020 Research and Innovation Action Bergamot, which has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 825303. Additional support was provided by Intel Corporation.

Lastly, I would like to thank my principal supervisor (Kenneth Heafield), my second supervisor (Rico Sennrich), my examiners (Barry Haddow and Graham Neubig), my thesis reviewers (Nikolay Bogoychev, Clara Vania, Kemal Maulana Kurniawan, Radityo Eko Prajoso), and my ILCC colleagues. I would also like to thank my family and my friends for their support.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(Alham Fikri Aji)
26 June 2020

Table of Contents

Chapter 1

Introduction

1.1 Motivation

Machine translation is the automatic process of translating texts from a source language (e.g., English) to another target language (e.g., German). Over the years, neural machine translation (NMT) (Ñeco and Forcada, 1996; Bahdanau et al., 2014) has become the state-of-the-art approach for machine translation and has been shown to outperform the older statistical machine translation approach (Sennrich et al., 2016a).

NMT models are typically resource-demanding, consisting of tens to hundreds of millions of parameters (Britz et al., 2017; Huang et al., 2019). Furthermore, training the model can take days to even weeks. For example, the winning system at the Workshop of Machine Translation (WMT) 2016 shared task for news translation was trained for 3 weeks on a single GPU (Sennrich et al., 2016a).[1] Similarly, Microsoft's English-to-German translation system at the WMT 2019 was trained for 4 days over 8 GPUs (Junczys-Dowmunt, 2019). Additionally, researchers and practitioners must often run multiple experiments, thus scaling the actual time and financial costs of producing an NMT model even more. Apart from the training phase, the deployment of the NMT model is also challenging. In an offline-based translation system, models must be distributed over the network to the client, which can be costly for large NMT models. For example, the size of the standard Transformer (Vaswani et al., 2017) model (a state-of-the-art NMT architecture) is approximately 300MB, depending on the vocabulary. Additionally, the model must be stored locally, with download and storage sizes increasing with the number of models required.

Notably, distributed training can improve training speed (Raina et al., 2009; Dean

[1] For an English-to-Czech model. The training times for other models were not reported

et al., 2012). One paradigm in distributed training is data parallelism, wherein the model is copied across workers and each worker trains based on different subsets of the training data. This training mechanism requires workers to exchange gradients, which is expensive since the gradient is as large as the model. Distributed NMT training involves a considerable amount of time being spent on communicating the gradients. In a four-node parallel training, ≈33% of the time is spent on communication. Ultimately, we only observe an ≈2.4x raw speed increase in this four-node parallel training over a single node. Therefore, the current distributed training is inefficient since the speed improvement is not linear with the cost (i.e., GPU hours). Our number is based on a 40-gigabit ethernet connection, while the improvement will be lower with slower consumer-grade hardware.

Prior work optimised parallel training by compressing the network traffic via pruning (Zhang et al., 2016; Lin et al., 2018), quantisation (Seide et al., 2014) or simply reducing gradient exchange frequency (Konečnỳ et al., 2016; Bogoychev et al., 2018). However, these lossy gradient compressions introduce noise and might be harmful to the model's quality. This issue is more problematic since the current NMT state-of-the-art architecture, Transformer (Vaswani et al., 2017) is highly sensitive to training conditions. Transformer has been reported to break when trained under noisy environments such as stale gradient updates (Chen et al., 2018; Ott et al., 2018) or incorrect learning-rate scheduling (Popel and Bojar, 2018; Nguyen and Salazar, 2019). Therefore, to make effective distributed training in Transformer, we must first understand what breaks Transformer and then design a low-communication cost distributed training that does not sacrifice translation quality.

After being trained, NMT models are deployed for use. In the case of a client-based translation service[2], models must be deployed locally. This case is useful in situations where data confidentiality is required, such as in government, corporate, or private document in general. Likewise, an offline translator will help users with limited internet access. Building an offline translation system means that the models must be distributed over the network and locally stored by the client. Therefore, the large NMT model introduces a challenge since it is costly to distribute and store. Moreover, the model must be locally loaded and executed for the translation; thus, it must fit a wide range of client computing resources (i.e., RAM).

Generally, model performance increases with the number of parameters (Huang et al., 2019). They show that model performance (measured in BLEU) with 4x more

[2] `https://browser.mt/`

parameters increases by more than two points. Therefore, naively reducing the number of parameters (i.e., with fewer layers or unit size) may reduce quality. Model compression techniques such as quantisation can be applied as an alternative. Hubara et al. (2016); Quinn and Ballesteros (2018); Jacob et al. (2018) have shown that neural network models can be represented with lower bit precision, thus requiring less storage space without sacrificing quality. However, it has been demonstrated that compressing NMT models tends to be more challenging when compared to convolutional neural network (CNN)-based models, which are often used in computer vision. Hubara et al. (2016) can quantise CNN models to 1-bit precision (32x smaller in size), whereas Quinn and Ballesteros (2018); Jacob et al. (2018) can only quantise Transformer up to 8-bit precision (4x smaller in size) before exhibiting quality degradation.

This thesis focuses on methods aimed at optimising the training and deployment of NMT models. We hypothesise that we can improve NMT training efficiency by introducing some approximations. We explore methods of gradient and model compression for cheaper network communication cost and reduced memory consumption. This thesis also explores methods of training the model under such compression. We also argue that different NMT architectures have different sensitivities towards noisy training; therefore, we will contrast both recurrent neural network (RNN)- and Transformer-based models in most of our experiments.

1.2 Thesis Structure

This thesis demonstrates that we can train NMT models under different approximations to improve efficiency. We approximate gradients by setting small values to zero to reduce network traffic. Similarly, we approximate the model with quantisation, thus reducing the download size. Since these approximations introduce noise, we minimise it by using approximation errors as feedback through the utilisation of local gradients and model pre-training. The remainder of this thesis will be structured as follows.

- **Chapter 2** provides a brief theoretical background. We first discuss the architectures of neural machine translation used in this thesis as well as some of the distributed training paradigms.

- **Chapter 3** explores asynchronous Transformer training. We determine that the Transformer model cannot be trained asynchronously. We then blur the lines

between asynchronous and synchronous training and demonstrate that stale gradients are the main cause of the sub-par performance of asynchronous Transformers. Effective batch sizes in asynchronous training are also smaller, which negatively affects the performance, but not as much. We also show that the RNN-based model is more robust against such noise. Furthermore, we explore a solution involving mimicking the behaviour of synchronous training while maintaining the asynchronous speed by accumulating the gradients server-side to reduce the average staleness while increasing the effective batch size.

- **Chapter 4** describes and explores a method of reducing the communication cost in distributed training by only exchanging the most significant gradients (in absolute value). This sparsification approach is designed based on our finding that gradient distribution is skewed (most of them are near zero). The compression errors from pruning small gradients add up; therefore, we must store the errors instead of discarding them, then add them to the next update step (dubbed as an error feedback mechanism). We demonstrate that we can ignore 99% of the gradients without significantly affecting the quality. With a simple value-index encoding, we reduce the network cost by a factor of 50.

- **Chapter 5** addresses the issue of the gradient sparsification technique discussed in Chapter 4. Since the gradients are compressed, the updates become noisy. Furthermore, the error feedback mechanism introduces stale gradients. These problems result in slower convergence; as such, the model requires more data to reach the same quality and can even break the Transformer model completely. This chapter discusses the idea of reconstructing compressed gradients by utilising local gradients. We demonstrate that adding local gradients can improve the convergence without sacrificing the reduced communication costs of sparse gradient updates. We also explore distributed training under a multi-node scenario wherein communication cost is more expensive.

- **Chapter 6** discusses cross-lingual transfer learning as a method of pre-training an NMT model. We measure the translation quality impact to gain a black box understanding of the transferred information. We determined that transfer learning improves the performance of low-resource NMT regardless of the language or the embedding configurations. We also found that models trained from a pre-trained high-resource model converge faster than the one trained from the beginning, despite not showing final quality improvement. Furthermore, we also

observe that transfer learning can eliminate the warm-up phase in Transformer training, which can further hasten the convergence in high-resource NMT.

- **Chapter 7** explores reducing the memory size of NMT models by quantising the parameters. We determined that the parameter distribution in the NMT model is not uniform, with most of them are close to zero. Therefore, the uniformly distributed fixed-point quantisation commonly used in model compression is not suitable. This chapter explores logarithmic-based quantisation to enable more quantisation centres to represent smaller values. We find that the model must be trained in full 32-bit precision first before switching to a lower bit. Similar to gradient sparsification in Chapters 3 and 4, the quantisation error is kept as an error feedback since the model converged poorly otherwise. We can compress the model up to 4-bit precision with insignificant quality damage.

- **Chapter 8** summarises and concludes the thesis. We also discuss ideas for possible future work.

1.3 Contributions

We make the following contributions:

- Experiments demonstrating that noisy gradient updates–mainly from staleness–can result in Transformer models being unable to be trained asynchronously. This work is based on Aji and Heafield (2019a).

- A method to reduce the communication cost in data parallelism training by only exchanging large gradients. We further improve this method by incorporating local gradients to restore the quality of compressed gradients. This work is based on Aji and Heafield (2017) and Aji et al. (2019).

- Black box experiments on transfer learning in an attempt to understand what makes transfer learning work, as well as its application as a model initialisation for faster and more stable training. This work is based on Aji et al. (2020).

- A method to compress and re-train the model under a log-based quantisation. This work is based on Aji and Heafield (2019b).

Chapter 2

Background

This chapter briefly explains the concept of neural machine translation and methods to train neural machine translation. It also discusses best practices and challenges in training neural machine translation, especially in parallel settings.

2.1 Neural Machine Translation

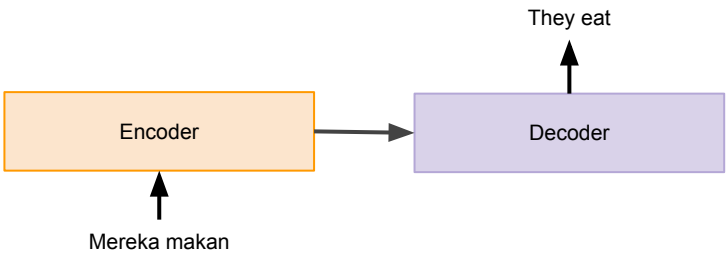


Figure 2.1: NMT with an encoder-decoder architecture. The input is first encoded in the encoder layer(s). The output is then generated by the decoder layer(s).

Machine translation is an automatic process of translating a text from a source language to another target language (e.g., English to German). Current NMT systems are often based on encoder-decoder architecture (Sutskever et al., 2014; Cho et al., 2014). From a high-level perspective, the architecture works as follows. First, the encoder processes a given source sequence. Then, the encoder passes the processed information to the decoder, which generates a corresponding target sequence. This flow is illustrated in Figure 2.1. In this section, we describe how to represent the source and target sequences. We also discuss the two common architectures for the

encoder and decoder layer: RNN (Cho et al., 2014) and Transformer (Vaswani et al., 2017).

2.1.1 Training Objective

Given a source sequence s, an NMT model with parameters θ produces a target t with a certain probability of $P(t|s,\theta)$. NMT training attempts to maximise this probability, given the training data as pairs of source and target sequences S and T. Alternatively, NMT minimises loss, which is often computed as a negative log-likelihood, as follows:

$$L(S,T,\theta) = -\frac{1}{N}\sum^{N} log(P(t_i|s_i,\theta)) \tag{2.1}$$

The parameters are often randomly initialised. We train the model by updating the parameters with the derivative of loss with respect to each parameter weight.

$$\theta = \theta - \alpha\frac{dL(S,T,\theta)}{d\theta} \tag{2.2}$$

Where α is a learning rate, a multiplier is used to scale the movement. Since the training size is often quite large, the loss is approximated from a subset of the training examples (mini-batches). A more detailed discussion on learning rate and mini-batches is presented in Subsection 2.1.4.

Previous work has attempted to initialise the parameters with weights trained on a similar task with the aim of transferring some knowledge (Zoph et al., 2016). We explore more about this model transfer as a method to achieve improved initialisation in Chapter 6.

2.1.2 Sequence Representation

Typically, both source and target sentences are in the form of plain text or a string. Tokenisation is used to split such strings into a sequence of representative tokens, such as into words or characters. These sequences of tokens are then represented as vectors, which will be passed to the encoder or decoder so that they can be represented numerically. One naive way to represent the word is through a one-hot vector. The values in this vector are all 0, except one, which corresponds to the token's ID (set to 1). The one-hot vector is inefficient since its size scales with the vocabulary size. Hence, a more efficient word embedding vector is used. In this case, each word is represented

by a smaller trainable vector. Moreover, embedding vector enables the model to train a more meaningful representation for each token.

Each token in the data vocabulary is represented as an embedding vector. As a result, the embedding matrices (a collection of embedding vectors) are the largest trainable parameter. For example, assuming we have 50,000 unique tokens in the training set and given a common embedding vector size of 512 each, we would require 25M parameters (100MB) for the embedding alone. The parameter size can be reduced with tied embedding (Junczys-Dowmunt et al., 2018), which involves sharing the embedding vector between the encoder and decoder. Throughout this thesis, we adopt tied embedding unless stated otherwise.

With word-level tokenisation, the vocabulary is limited to the training set. Therefore, an NMT model trained under such a setting cannot represent a new word unseen in the training set. Tokenising the input into character level solves the vocabulary issue (assuming under the same character set) but makes the input sequence extremely long. One solution is to tokenise the sentences into the sub-word level with byte pair encoding (BPE) (Sennrich et al., 2016c; Gage, 1994), which is utilised in every experiment within this thesis. Under BPE, texts are tokenised into common sub-words by applying the following procedures:

1. We start by tokenising the sentences into characters.
2. Find the most frequent adjacent pair of tokens in the data and merge them into a new token. This process is called a merging operation.
3. Repeat step 2 for several iterations.

BPE breaks down rare words into subwords (in the worst case, to character level); therefore, the model is robust towards unseen words. Likewise, BPE represents frequent subwords in a single token, thereby keeping the sequence length reasonably short, compared to a character-based tokens.

2.1.3 Model Architectures

RNN-based Encoder and Decoder

The input sequence length in machine translation (and natural language processing tasks in general) is often dynamic. A recurrent neural network (RNN) is designed to process such a sequence with arbitrary length by having a loop mechanism that allows

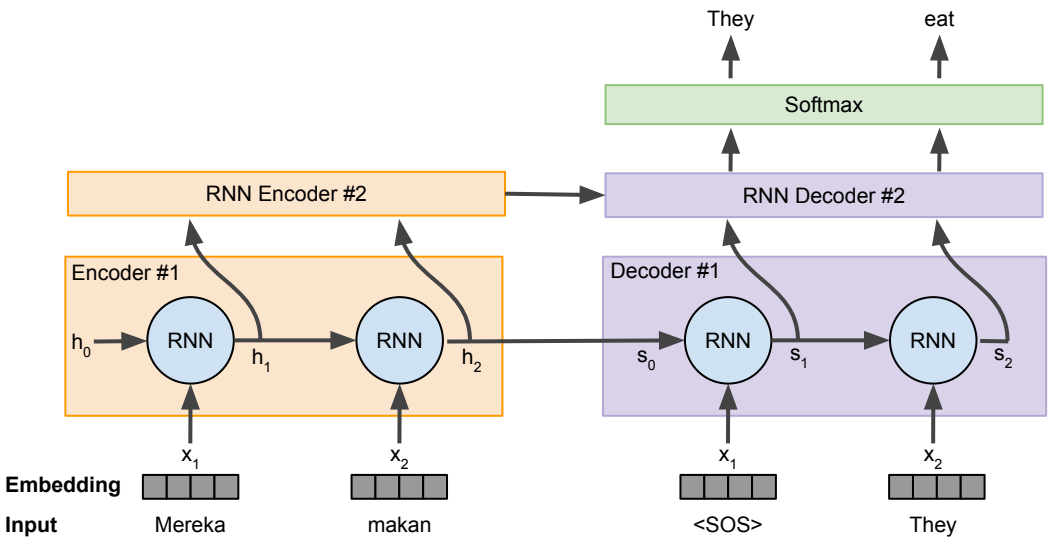


Figure 2.2: An Illustration of an RNN-based encoder and decoder architecture.

information to be processed step by step while maintaining a self-internal state. At time-step t, the RNN cell takes the $t-$th input $\mathbf{x}_t$ and its previous step internal state (also known as hidden state) $\mathbf{h}_{t-1}$ to update its hidden state to $\mathbf{h}_t$. Figure 2.2 illustrates the unrolled RNN-based encoder and decoder. An RNN updates the state, as follows:

$$\mathbf{h}_t = f(\mathbf{W}_h\mathbf{h}_{t-1} + \mathbf{W}_x\mathbf{x}_t + \mathbf{b}) \tag{2.3}$$

Function f is a non-linear activation function, such as $tanh$. $\mathbf{W}_h$ is a trainable weight matrix. Usually, we can also add a trainable bias $\mathbf{b}$. The produced hidden state is also be used as the cell's output, which can be passed on as the next layer's input.

The RNN decoder takes the encoder's final hidden state as its initial hidden state. The decoder then passes the hidden state to a softmax layer for prediction. The predicted output is then used as the input for the next time step.

RNNs suffer from vanishing gradients, which is especially common when the RNN is given a longer sentence as input. To mitigate this problem, LSTM Hochreiter and Schmidhuber (1997) is proposed as an alternative cell architecture. In LSTM, we introduce additional information known as the cell state. Similarly, at time step t, an LSTM cell takes the $t-$th input $\mathbf{x}_t$ and its previous internal states: hidden state $\mathbf{h}_{t-1}$ and cell state $\mathbf{c}_{t-1}$. LSTM utilises input gate $\mathbf{i}_t$, forget gate $\mathbf{f}_t$, and output gate $\mathbf{o}_t$, which are computed with an identical function under different trainable weights, as follows:

$$\mathbf{i}_t = \sigma(\mathbf{W}_i\mathbf{h}_{t-1} + \mathbf{U}_i\mathbf{x}_t + \mathbf{b}_i) \tag{2.4}$$

$$\mathbf{f}_t = \sigma(\mathbf{W}_f\mathbf{h}_{t-1} + \mathbf{U}_f\mathbf{x}_t + \mathbf{b}_f) \tag{2.5}$$

$$\mathbf{o}_t = \sigma(\mathbf{W}_o\mathbf{h}_{t-1} + \mathbf{U}_o\mathbf{x}_t + \mathbf{b}_o) \tag{2.6}$$

The LSTM employs a sigmoid function on these gates, forcing the value range to $[0, 1]$. Therefore, we can multiply them element-wise to gate the information flow.

A candidate state is computed as follows:

$$\bar{\mathbf{c}}_t = tanh(\mathbf{W}_c\mathbf{h}_{t-1} + \mathbf{U}_c\mathbf{x}_t + \mathbf{b}_c) \tag{2.7}$$

The cell state $\mathbf{c}_t$ is updated as the sum of the candidate state weighted by the forget gate and the previous cell state weighted by the input gate, as follows ($\odot$ denotes element-wise multiplication):

$$\mathbf{c}_t = \bar{\mathbf{c}}_t \odot \mathbf{i}_t + \mathbf{c}_{t-1} \odot \mathbf{f}_t \tag{2.8}$$

Lastly, we compute the output hidden state based on the cell state weighted by the output gate:

$$\mathbf{h}_t = tanh(c_t) \odot \mathbf{o}_t \tag{2.9}$$

LSTM only considers the information from the previous steps. A bidirectional LSTM is designed to overcome this limitation by having two layers of LSTMs–one with a reversed direction on top of another one. The output hidden states from both directional LSTMs are concatenated, before passing it to the next layer.

Attention Mechanism

A basic RNN-based NMT model initialises the decoder's initial hidden and cell state from the encoder's last state. This is the only way to transfer the information from the encoder to the decoder, which might not be sufficient to represent the entire sequence. Thus, NMT performance often degrades as the sequence length increases (Bahdanau et al., 2014). The attention mechanism is designed to facilitate more information flow to the decoder by allowing the decoder to gather the encoder's hidden states at different time steps (Bahdanau et al., 2014).

The attention function takes the current-step decoder hidden state $\mathbf{s}_t$ as a query and the encoder hidden states across each time step $\mathbf{h}_1, ..., \mathbf{h}_N$ as the keys. The attention computes the context vector $\mathbf{c}_t$ as the weighted sum of the keys, where the weight assigned is scored from the query to the corresponding key, as follows:

$$\mathbf{c}_t = \sum_i Softmax(S(\mathbf{s}_t, \mathbf{h}_i))\mathbf{h}_i \tag{2.10}$$

A softmax function is used to normalise the query-key score function S into a probability distribution:

$$Softmax(S(\mathbf{s}_t, \mathbf{h}_i)) = \frac{exp(S(\mathbf{s}_t, \mathbf{h}_i))}{\sum_j exp(S(\mathbf{s}_t, \mathbf{h}_j))} \tag{2.11}$$

Bahdanau et al. (2014) proposed the additive scoring function as follows:

$$S(\mathbf{s}_t, \mathbf{h}_i) = \mathbf{v}^T \, tanh(\mathbf{W}_Q \mathbf{s}_t + \mathbf{W}_K \mathbf{h}_i) \tag{2.12}$$

where $\mathbf{v}^T$, $\mathbf{W}_Q$ and $\mathbf{W}_K$ are all trainable parameters.

Transformer-based Encoder and Decoder

The Transformer architecture completely discards recurrent connections and focuses on more parallelisable attention mechanisms (Vaswani et al., 2017). In Transformer, the encoder layer consists of self-attention. This self-attention attends each token from a sequence to each token from the same sequence. The decoder works similarly, with an extra attention mechanism on top of the self-attention mechanism. This attention attends to the encoder, similar to the attention in RNN. The attention is followed by a feed-forward network. Each sub-layer (attention and feed-forward) is followed by an addition from a residual connection and a layer normalisation step. Since the Transformer has no recurrency, pre-determined positional encoding vectors are added to the embedding to inject sequence order information. An illustration of the Transformer architecture is shown in Figure 2.3. Vaswani et al. (2017) used a scaled dot product to compute the attention score, as follows:

$$S(\mathbf{q}_t, \mathbf{k}_i) = \frac{\mathbf{W}_Q \mathbf{q}_t \cdot \mathbf{W}_K \mathbf{k}_i}{\sqrt{d_k}} \tag{2.13}$$

A score of the query vector $\mathbf{q}_t$ and a key $\mathbf{k}_i$ is obtained by applying dot products after multiplying them with trainable matrices $\mathbf{W}_Q$ and $\mathbf{W}_k$. For the self-attention in encoders, both the query vector $\mathbf{q}_t$ and the keys $\mathbf{k}_i$ are obtained from the output of the

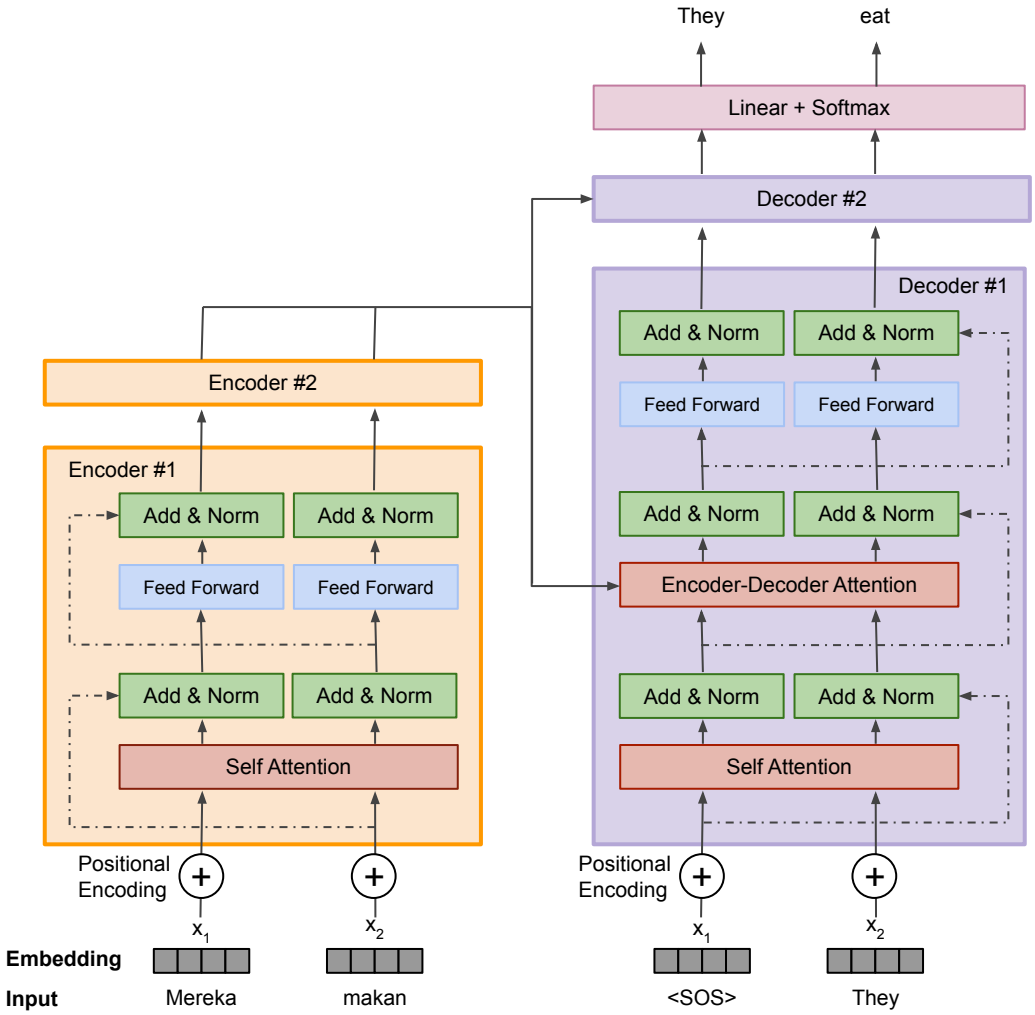


Figure 2.3: An Illustration of a Transformer-based encoder and decoder architecture.

previous encoder layer. Regarding the encoder-decoder attention in the decoder, the query originates from the previous decoder layer, while the keys originate from the output of the final encoder layer. We scale down the score based on the dimension of the key d_k.

The attention output c_t computes a weighted sum of value vectors, which can be obtained by multiplying the keys with a trainable parameter $\mathbf{W}_V$, as follows:

$$\mathbf{c}_t = \sum_i Softmax(S(\mathbf{q}_t, \mathbf{k}_i))\mathbf{W}_V \ \mathbf{k}_i \tag{2.14}$$

Transformer also employs a multi-head attention mechanism. Therefore, Transformer has N (usually 8) independent attentions under separate attention weights outputting different weighted sum. The output from each head is then concatenated.

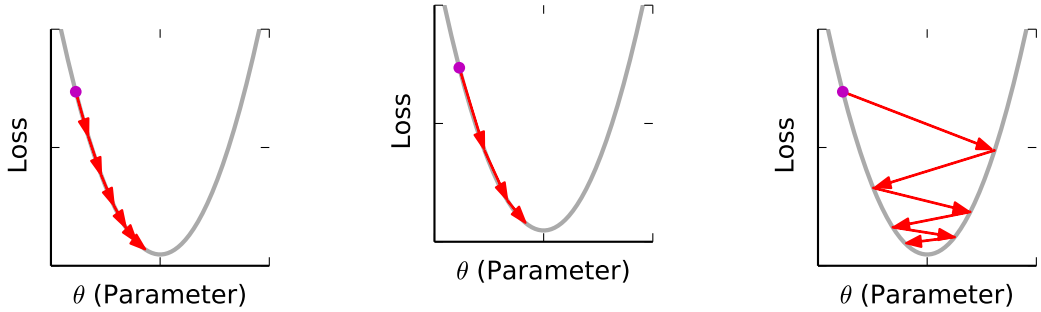
The attention mechanism in the encoder only requires information from the previous layer. Thus, a Transformer-based encoder is more parallelisable compared to the sequential RNN architecture. Therefore, the Transformer can process mini-batches faster than the RNN, which we empirically report in our experiments involving both architectures (Chapter 3 and Chapter 5).

2.1.4 Training Practices

Parameters in the NMT model are optimised with stochastic gradient descent. Training is done in mini-batches, which represent a small subset of the training data. We run a forward pass through the model to obtain the prediction and error, and then run the back-propagation to compute the gradient. Then, we update the parameters by subtracting them with the combined gradients from each sentence.

This forward and backward pass, followed by the parameter update is repeated multiple times (i.e., multiple mini-batches or steps) until some arbitrary stopping condition is achieved. A common stopping condition involves training the model for a pre-determined amount of steps or epochs (multiple passes of the entire training set). Additionally, early stopping criterion can be added. For example, they can be added to stop training if the model does not improve after several evaluations of the validation set.

When comparing the training speed, simply measuring the time required to finish the training may not be sufficient since training duration might not capture the convergence speed. For example, we can set a model to train for ten epochs, while the model only requires two epochs to reach its best performance in practice. The early stopping criterion is somewhat useful to avoid wasting computational resources, as in that case. However, it is also unstable since the early stopping criterion is often reset by negligible improvement (e.g., assuming we stop the training after five validations with no improvement). If we gain a negligible and potentially noisy improvement in training loss after the fourth validation, we must restart the counting back to 0. Similarly, measuring the time required to reach the best validation score is very unstable. Therefore, in all of our experiments involving measuring the training speed, we measure the time required to reach a certain near-convergence quality threshold.



(a) With a low learning rate, more updates are required to reach the local minima.

(b) With the optimal learning rate, parameter reaches the local minima with less updates.

(c) Parameter "jumps" around the search space if the learning rate is too high.

Figure 2.4: Illustrations of parameter movements across different learning rates.

Learning Rate

The gradient is multiplied by a learning rate to adjust the movement of the parameter. A higher learning rate means that the parameters move further for each update. When the learning rate is low, the parameter moves slowly. Therefore, the training is more stable but may take more time to complete. In contrast, the parameter moves faster with a higher learning rate. However, if the learning rate is too high, the parameter moves too far and might cause the model to overshoot. The training can even diverge if the learning is extremely high. Illustrations of these cases are shown in Figure 2.4.

Since this thesis focuses on efficient training, choosing the right learning rate is important. We optimise the learning rate by performing a grid search. Then, we choose the highest learning rate that retains the final quality while also reaching such quality the fastest.

Training might be unstable at the early stage of the training. We can employ a learning rate warm-up by using a lower learning rate at the beginning of the training and periodically increasing it throughout the training. Learning rate warm-up has shown to be important for training Transformers (Popel and Bojar, 2018). In Chapter 6, we also explore training Transformers without warm-up by employing a pre-trained model.

Learning rate warm-up makes the training more stable by scaling down the learning rate, thereby essentially slowing down the convergence. Hence, where learning rate warm-up is applied, we search the minimum warm-up period that does not degrade the model's quality and convergence.

Adaptive Learning Rate

Kingma and Ba (2014) proposed the Adam optimiser, which independently adjusts individual learning rates for different parameters. The Adam optimiser has shown to improve training speed and performance (Kingma and Ba, 2014), and is commonly used in neural machine translation experiments (Sennrich et al., 2016a; Bojar et al., 2017, 2018).

Adam estimates the full gradient with an exponentially decaying average m_t of gradients g_t:

$$m_t \leftarrow \beta_1 m_{t-1} + (1-\beta_1) g_t \tag{2.15}$$

where β_1 is a decay hyperparameter. It also computes a decaying average v_t of second moments:

$$v_t \leftarrow \beta_2 v_{t-1} + (1-\beta_2) g_t^2 \tag{2.16}$$

where β_2 is a separate decay hyperparameter. The squaring g_t^2 is taken element-wise. These estimates are biased because the decaying averages were initialised to zero. Adam corrects for the bias to obtain unbiased estimates $\hat{m}_t$ and $\hat{v}_t$:

$$\begin{aligned} \hat{m}_t &\leftarrow m_t / (1-\beta_1^t) \\ \hat{v}_t &\leftarrow v_t / (1-\beta_2^t) \end{aligned} \tag{2.17}$$

These estimates are used to update parameters θ, as follows:

$$\theta_t \leftarrow \theta_{t-1} - \alpha \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \varepsilon} \tag{2.18}$$

where α is the learning rate hyperparameter and ε prevents element-wise division by zero.

We use the Adam optimiser in all of our NMT experiments throughout this thesis. In Chapter 3, we investigate the behaviour of the Adam optimiser under noisy and stale gradients found in asynchronous training.

Batch Size

The gradients of a single mini-batch are the combination (usually by averaging or summing) of the individual gradient of each training example. According to Smith and Le

(2017), gradient noise scales down as the batch size increases since we are averaging more samples. Popel and Bojar (2018); Ott et al. (2018) empirically show that NMT models converge better with larger batch size. We also explore this behaviour while comparing synchronous and asynchronous training since the former has a larger effective batch size. We discuss this more in Chapter 3. As a rule of thumb, we always maximise the batch size (with regards to the GPU limitation) in our experiments throughout this thesis.

2.2 Distributed Training

The NMT model can be trained in parallel by distributing the training across workers (i.e., GPUs or CPUs). In this section, we discuss different paradigms and methods regarding distributed training.

2.2.1 Model vs Data Parallelism

In model parallelism, the model is divided and distributed across workers. Each worker is responsible for performing forward and backward passes on its own subset model and then passing the output to the next worker. The main objective of this parallelism is to reduce the memory consumption required to store the model and to enable the training of larger models (Huang et al., 2019). However, model parallelism does not necessarily improve the training speed, as the computation is performed sequentially layer-by-layer, similar to non-parallel training. Additionally, model parallelism also requires sending some information across workers, which costs additional time.

In data parallelism, the model is copied and distributed across workers. The training data is split across workers, where each worker trains independently based on its data subset. For each batch, each worker informs the computed gradients to the parameter server: a node that stores the parameter. The parameter server then processes the gradients and performs the parameter updates. Then, each worker calibrates its local model with the newest one from the parameter server.

Assuming synchronous training, the gradients are combined in the parameter server; thus, data parallelism behaves as if we increased the effective batch size. Notably, increasing the batch size is always typically preferred over using the data parallelism—if possible. For example, it is inefficient to train with two workers with a 16-batch size each if a single worker with a 32-batch size is possible. While both cases have an ef-

fective batch size of 32, the former uses double the GPU cost of the latter. Moreover, a GPU benefits from parallel computation; thus, processing a batch of size 32 is not usually twice as costly as processing a batch of size 16. The communication overhead also contributes to inefficiency. We can also observe increasing batch size per GPU as a method of reducing the communication frequency, thereby increasing training efficiency. Empirically, with a typical Transformer setting, 2x16 GPUs process batches 10% slower than a 1x32 GPU.

With data parallelism, we can process more sentences in single mini-batch to improve the training speed. However, in practice, each worker must communicate the gradients to and from the server per batch, which is as large as the model and significantly reduces the efficiency of this method. We address this issue in Chapter 4 and Chapter 5.

2.2.2 Synchronous vs. Asynchronous Training

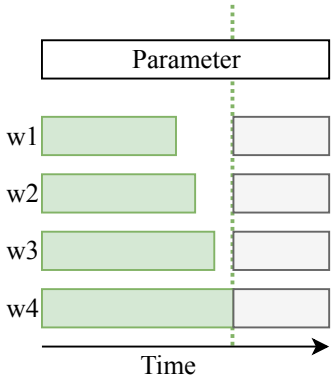
Parallel training can be either synchronous or asynchronous. In synchronous training (usually the default data parallelism behaviour), the parameter server waits for each of the workers to send the gradients. The server then combines the gradients and applies the parameter update. Then, the server broadcasts the newly updated model[1]. One problem with synchronous training is that each worker must wait for the slowest worker, thus leaving some of the faster workers to idle.

In contrast, asynchronous training updates the parameter directly after obtaining the gradients. Likewise, the model also fetches the recent parameter in arbitrary time– usually after sending the gradients; therefore, workers do not idle. However, this behaviour introduces a stale update (i.e., a gradient update computed from an outdated parameter version). Moreover, the mini-batch size is smaller than the synchronous counterpart since no gradient summing is involved. We explore the difference between synchronous and asynchronous training in more detail and discuss the issue of stale gradients in Chapter 3.

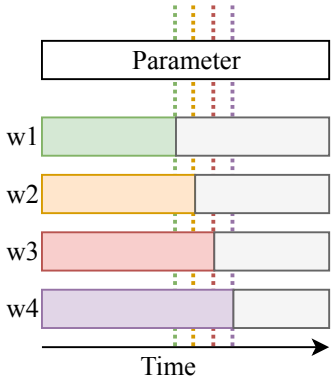
2.2.3 Parameter Sharding

Let D be the number of parameters and N be the number of nodes. Intuitively, each worker needs to send D values (local gradient) to the server and receive D values (up-

[1] On a technical level, we can also broadcast the summed gradients alone and let the workers update their model locally



(a) Workers are idle in synchronous training as they wait for other workers to finish before updating the parameter (vertical line).



(b) Parameter updates are immediate in asynchronous training; therefore, no idle workers but may cause stale updates.

Figure 2.5: Synchronous vs asynchronous training.

dated parameter or combined gradient) from the server. Naively, we can assign a node to be the server where each worker communicates to the server. Thus, the server receives $(N-1)*D$ values, assuming we assign one of the workers to be the server. This approach is expensive since communication bandwidth per device is limited, which can cause a single bottleneck.

To avoid this issue, Dean et al. (2012) proposed a distributed stochastic gradient descent (SGD) with parameter sharding. Dean et al. (2012) divide the data into N equal-sized shards and distribute them to N different servers for the communication load to be balanced. These servers are also jointly located with the workers. Hence, each worker becomes both a client and a server and is responsible for different $1/N$-th of the parameters.

Clients have a copy of all parameters, which they use to compute gradients. These gradients are split into N pieces and pushed to the appropriate servers. Similarly, each

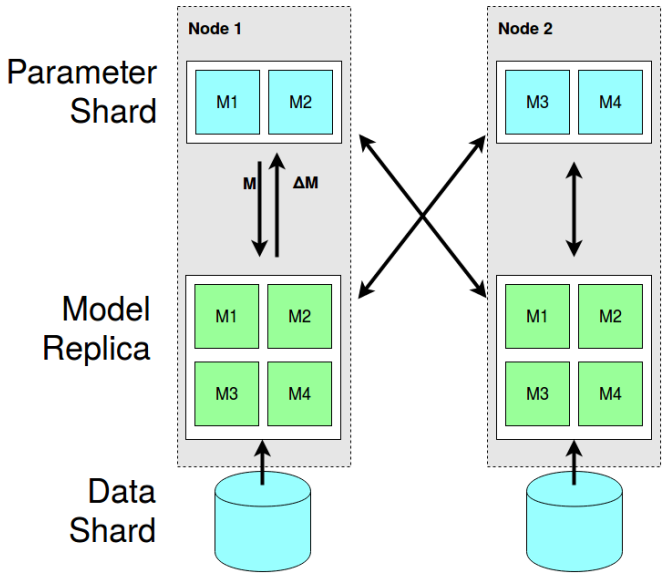


Figure 2.6: Data-parallelism architecture with parameter sharding.

client pulls parameters or summed gradients from all servers.

Using this communication mechanism, each worker will communicate D/N of parameters to N different server shards, thus resulting in a constant D bandwidth cost. Parameter pulling follows the same communication, so the total memory sent by a worker is $2D$. This bandwidth cost is constant regardless of the number of workers.

2.2.4 Scaling the Number of Workers

Using more workers (GPUs or nodes) does not necessarily imply faster training. Assuming synchronous training, more workers equals a larger batch size. If we assume instantaneous communication, each mini-batch requires the same processing time. Therefore, scaling the workers by N means that we potentially process the data N times faster. Ideally, we would like the model to achieve the same quality after seeing the same amount of data when increasing the batch size to achieve linear improvements in training time. With this assumption, if we can process the data N times faster, we should also reach the same model quality N times faster.

If the gradient is not scaled (e.g., from a scale-invariant optimiser such as the Adam optimiser (Kingma and Ba, 2014)), the update magnitude will be the same and the parameter will move at the same rate regardless of the batch size. This behaviour makes training with a larger batch size less data-efficient since we consume more training examples per batch. Therefore, without additional tuning, we cannot achieve the same quality measurement with the same amount of data since we have fewer updates.

Recent studies (Goyal et al., 2017; Popel and Bojar, 2018; Ott et al., 2018) suggest scaling the learning rate linearly with the batch size to increase the training efficiency in larger batch sizes. This heuristic makes sense since, given the same amount of data, we only perform updates N times less often if we have an N times larger batch size. Assuming the same movement magnitude, we would like to move N times further per update to catch up with the lower batch size setting. Smith and Le (2017) mentioned that gradient noise is proportional to the learning rate divided by the batch size. Therefore, increasing batch size is a form of reducing gradient noise and can mitigate the noise introduced from a higher learning rate.

Similarly, the learning rate warm-up must be adjusted correspondingly. If the warm-up period is defined by the number of steps, it must be scaled down proportionally since we aim to achieve the same scaled learning rate given the same epochs.

In practice, achieving linear speed increase with more workers (thus larger batch size) is challenging since we would have to accomplish the same convergence with significantly fewer updates; therefore, we scale the learning rate and learning rate warm-up. However, Goyal et al. (2017) reported that the learning rate could not be scaled indefinitely since the model will be untrainable when the learning rate is too high. Similarly, lowering the warm-up too much will result in the learning rate increment being too steep, especially when the maximum learning rate is very high. Therefore, both the learning rate and the warm-up must be adjusted sub-linearly, resulting to a sub-linear speed increase. To better illustrate this point, assuming a typical setting with around 100000 updates of training. If we scale the number of workers with a very large $N = 1000$, it is doubtful that the model will only need to make 10 updates with a massive learning rate to converge. We explore the scaling of workers up to 48 GPUs in Chapter 5.

Chapter 3

Asynchronous Transformer Training

This chapter explores training Transformer models in asynchronous stochastic gradient descent. We first investigate why asynchronous Transformers under-perform, and then apply a solution by mimicking synchronous training behaviour while maintaining asynchronous speed. This chapter is based on Aji and Heafield (2019a).

3.1 Introduction

Models based on Transformers (Vaswani et al., 2017) achieve state-of-the-art results in various machine translation tasks (Bojar et al., 2018). Distributed training is crucial to training these models in a reasonable amount of time, with the dominant paradigms being asynchronous or synchronous SGD. Prior work (Chen et al., 2016, 2018; Ott et al., 2018) has noted that asynchronous SGD yields low-quality models without elaborating further; we confirm this experimentally in Section 3.2.1. Rather than abandoning asynchronous SGD, we aim to repair its convergence.

To understand why asynchronous SGD under-performs, we first blur the lines between asynchronous and synchronous methods. Asynchronous and synchronous SGD have two key differences: batch size and staleness. Synchronous SGD increases the batch size in proportion to the number of processors because gradients are summed before applying an update. Asynchronous SGD updates with each gradient as it arises, resulting in the batch size being the same as on a single processor. Asynchronous SGD also has stale gradients because parameters may update several times while a gradient is being computed.

To separate the impact of batch size and stale gradients, we perform a series of experiments on both RNNs and Transformers by manipulating the batch size and in-

jecting staleness. Our experiments indicate that small batch sizes only slightly degrade quality, while stale gradients substantially degrade quality.

To restore convergence, we propose a hybrid method that computes gradients asynchronously, sums gradients as they arise and updates less often. Gradient summing has been applied to increase batch size or reduce communication (Dean et al., 2012; Lian et al., 2015; Ott et al., 2018; Bogoychev et al., 2018); moreover, we observe that it also reduces harmful staleness. In a sense, updating less often increases staleness because gradients are computed with respect to parameters that could have been updated. However, if staleness is measured by the number of intervening updates to the model, then staleness is reduced because updates occur less frequently. Empirically, this hybrid method converges comparably to synchronous SGD, preserves the final model quality and runs faster because processors are not idle.

3.2 Exploring Asynchronous SGD

3.2.1 Baseline: The Problem

To motivate this chapter and set baselines, we first measure how poorly Transformers perform when trained with baseline asynchronous SGD (Chen et al., 2016, 2018; Ott et al., 2018). We train a Transformer model under both synchronous and asynchronous SGD, contrasting the results with an RNN model. Moreover, we sweep learning rates to verify that this effect is not an artefact of choosing hyperparameters that favour one scenario. Further details on the experimental setup appear in Section 3.4.1.

The results presented in Table 3.1 confirm that asynchronous SGD generally yields lower-quality systems than synchronous SGD. For Transformers, the asynchronous results are catastrophic, often yielding 0 BLEU. We can also see that Transformers and asynchronous SGD are more sensitive to learning rates compared to RNNs and synchronous SGD.

To understand why asynchronous SGD under-performs, we run a series of ablation experiments based on the differences between synchronous and asynchronous SGD. We focus on two main aspects: batch size and stale gradient updates.

3.2.2 Batch Size

As previously discussed in Chapter 2, parallel synchronous training is essentially training with larger batch size since synchronous SGD sums gradients from all processors.

	Trans. BLEU		RNN BLEU	
Learn Rate	Sync.	Async.	Sync.	Async.
0.0002	35.08	13.27	34.11	33.77
0.0003	35.66	30.72	33.79	33.95
0.00045	35.59	5.21	33.68	33.68
0.0006	35.42	0.00	34.30	33.76
0.0009	34.79	0.00	34.28	33.47
0.0012	33.96	0.00	34.37	33.23
0.0024	29.35	0.00	33.98	32.83
0.00375	25.25	0.00	33.80	31.89

Table 3.1: Performance of the Transformer and RNN model trained synchronously and asynchronously, across different learning rates.

In asynchronous SGD, each update uses a gradient from one processor.

Using a larger batch size reduces noise in estimating the overall gradient (Wang et al., 2013; Smith and Le, 2017) and has been shown to slightly improve performance (Smith et al., 2017; Popel and Bojar, 2018). To investigate whether small batch sizes are the main issue with asynchronous Transformer training, we sweep batch sizes and compare the performance with synchronous training.

3.2.3 Gradient Staleness

We have introduced the gradient staleness issue in Chapter 2 and will discuss it in greater detail within this section. In asynchronous training, a computed gradient update is applied immediately to the model, without having to wait for other processors to finish. This approach may cause a stale gradient, where parameters have updated while a processor was computing its gradient. Staleness can be defined as the number of updates that occurred between the processor pulling parameters and pushing its gradient. In an ideal case where every processor spends equal time to process a batch, asynchronous SGD with N processors produces gradients with staleness $N-1$. That is, between the parameter pull and gradient push processes of a worker, the other $N-1$ are expected to send their gradients. Empirically, we can also expect an average staleness of $N-1$ with a normally distributed computation time (Zhang et al., 2016).

An alternative way to interpret staleness is the distance between the parameters with which the gradient was computed and the parameters were updated by the gradi-

ent. Therefore, a higher learning rate contributes to staleness as the parameters move faster.

Prior work has shown that neural models can still be trained on stale gradients, albeit with potentially slower convergence or lower quality. Furthermore, Zhang et al. (2016); Srinivasan et al. (2018) report that model performance degrades in proportion to the gradient staleness. We introduce artificial staleness to confirm the significance of gradient staleness towards Transformer performance.

3.3 Incremental Updates in Adam

Investigating the effect of batch size and staleness further, we analyse why it makes a difference whether gradients computed from the same parameters are applied one at a time (incurring staleness) instead of being summed then applied once (as in synchronous SGD). In standard stochastic gradient descent, there is no difference: gradients are multiplied by the learning rate and then subtracted from the parameters in either case. In practice, gradients reported by different processors are usually not the same: they are noisy estimates of the true gradient. Therefore, the Adam optimiser handles incremental updates and sums differently.

Notably, the Adam optimiser is scale-invariant. For example, suppose that two processors generate gradients 0.5 and 0.5 with respect to the same parameter in the first iteration. Incrementally updating with 0.5 and 0.5 is the same as updating with 1 and 1 due to scale invariance; thus, updating with the summed gradient, 1, will only move parameters half as far. This is the theory underlying the rule of thumb, which states that the learning rate should scale with batch size (Ott et al., 2018).

The Adam optimiser update parameters θ based on the estimations of first (m_t) and second moments (v_t) of gradients are as follows:

$$\theta_t \leftarrow \theta_{t-1} - \alpha \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \varepsilon} \tag{3.1}$$

where α is the learning rate hyperparameter and ε prevents element-wise division by zero.

Replacing estimators in the update rule with statistics they estimate and ignoring the usually-minor ε, we have:

$$\frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \varepsilon} \approx \frac{E g_t}{\sqrt{E(g_t^2)}} \tag{3.2}$$

Time (t)		0	1	2	3	4	5	6
Constant	g_t		1	1	1	1	1	1
	m_t	0	0.1	0.19	0.271	0.344	0.41	0.469
	v_t	0	0.02	0.04	0.059	0.078	0.096	0.114
	$\hat{m}_t$	0	1	1	1	1	1	1
	$\hat{v}_t$	0	1	1	1	1	1	1
	θ	0	-0.001	-0.002	-0.003	-0.004	-0.005	-0.006
Scaled	g_t		0.5	1.5	0.5	1.5	0.5	1.5
	m_t	0	0.05	0.195	0.226	0.353	0.368	0.481
	v_t	0	0.005	0.05	0.054	0.098	0.101	0.144
	$\hat{m}_t$	0	0.5	1.026	0.832	1.026	0.898	1.026
	$\hat{v}_t$	0	0.25	1.26	0.917	1.26	1.05	1.26
	θ	0	-0.001	-0.002	-0.003	-0.004	-0.005	-0.005
Different	g_t		-1	2	-1	2	-1	2
sign	m_t	0	-0.1	0.11	-0.001	0.199	0.079	0.271
	v_t	0	0.02	0.1	0.118	0.195	0.211	0.287
	$\hat{m}_t$	0	-1	0.579	-0.004	0.579	0.193	0.579
	$\hat{v}_t$	0	1	2.515	2	2.515	2.2	2.515
	θ	0	0.001	0.001	0.001	0.000	0.000	-0.000

Table 3.2: The Adam optimiser slows down when gradients have larger variance, even if they have the same average, in case 1. When alternating between -1 and 2, the Adam optimiser takes six steps before the parameter has the correct sign. Updates can even slow down if gradients point in the same direction but have different scales. The learning rate is $\alpha = 0.001$.

which expands following the variance identity

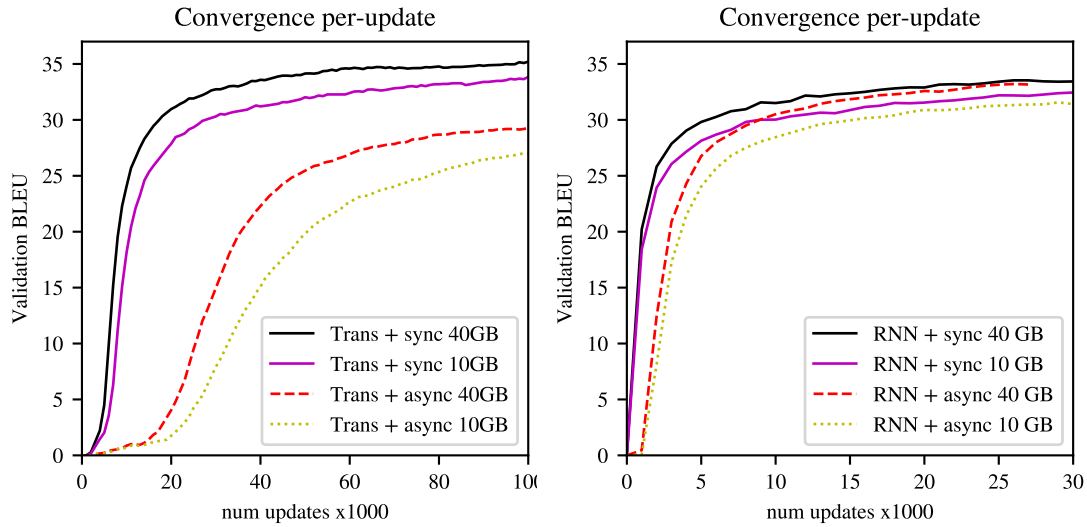
$$\frac{Eg_t}{\sqrt{E(g_t^2)}} = \frac{Eg_t}{\sqrt{Var(g_t) + (Eg_t)^2}} \tag{3.3}$$

Upon dividing both the numerator and denominator by $|Eg_t|$, we obtain:

$$= \frac{\text{sign}(Eg_t)}{\sqrt{Var(g_t)/(Eg_t)^2 + 1}} \tag{3.4}$$

The term $Var(g_t)/(Eg_t)^2$ is statistical efficiency, the square of the coefficient of variation. In other words, the Adam optimiser gives higher weight to gradients if historical samples have a lower coefficient of variation. The coefficient of variation of a sum of N independent[1] samples decrease as $1/\sqrt{N}$. Hence, sums (despite having less frequent updates) may cause the Adam optimiser to move faster since they have a smaller coefficient of variation. An example of this is presented in Table 3.2: updating with 1 moves faster than individually applying -1 and 2.

In Table 3.2, we present examples of noise causing the Adam optimiser to slow down. However, summing gradients smooths out some of the noise. Next, we examine the formal basis for this effect.



(a) Convergence in the Transformer model

(b) Convergence in the RNN Model

Figure 3.1: The effect of batch sizes on convergence over updates of Transformer and RNN models.

[1] Batch selection considers computing time, so noise is technically not independent.

3.4 Ablation Study

We conduct ablation experiments to investigate the poor performance in asynchronous Transformer training for the neural machine translation task.

3.4.1 Experiment Setup

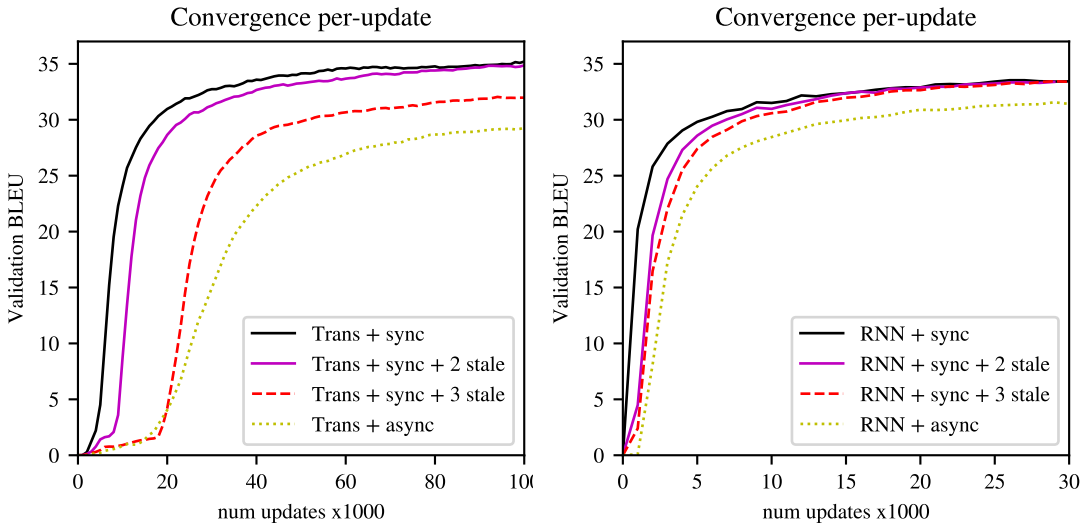
Our experiments use systems for the WMT 2017 English-to-German news translation task. Transformer comes standard with six encoders and six decoder layers Vaswani et al. (2017). The RNN model (Miceli-Barone et al., 2017) is based on the winning WMT17 submission (Sennrich et al., 2017) with eight layers. Both models use back-translated monolingual corpora (Sennrich et al., 2016b) and byte pair encoding (Sennrich et al., 2016c) with 36000 merging operations.

We follow the remaining hyperparameter settings on both Transformer and RNN models, as suggested in previous work (Vaswani et al., 2017; Sennrich et al., 2017). Both models were trained on four GPUs with a dynamic batch size of 10 GB per GPU using the Marian toolkit (Junczys-Dowmunt et al., 2018). Both models are trained for eight epochs or until reaching five continuous validations without loss improvement. Quality is measured on newstest2016 using sacreBLEU (Post, 2018), while preserving newstest2017 as a test for later experiments. Transformer's learning rate is linearly warmed up for 16k updates. We then apply an inverse square root learning rate decay following Vaswani et al. (2017) for both models. All of these experiments use the Adam optimiser, which has been shown to perform well on a variety of tasks (Kingma and Ba, 2014) and was used in the original Transformer publication (Vaswani et al., 2017).

For subsequent experiments, we will use a learning rate of 0.0003 for Transformers and 0.0006 for RNNs. These were near the top in both asynchronous and synchronous settings (Table 3.1).

3.4.2 Batch Size

We first explore the effect of batch size on the model's quality. We use dynamic batching, in which the toolkit fits as many sentences as it can into a fixed amount of memory (for example, more sentences will be in a batch if all of them are short). Hence, batch sizes are denominated in memory sizes. Our GPUs each have 10 GB available for batches which correspond to an average of 250 sentences.



(a) Transformer model with lr = 0.0003 (b) RNN model with lr = 0.0006

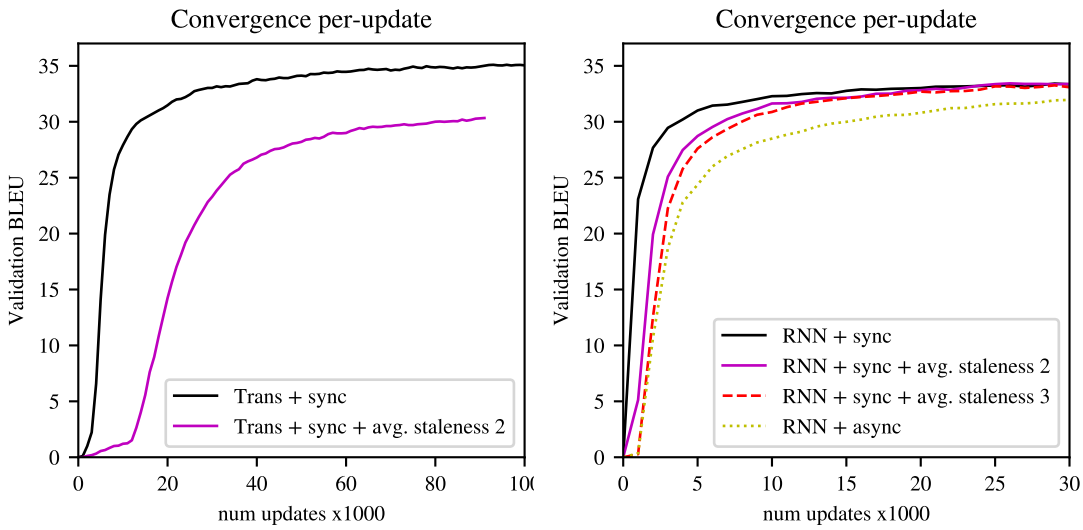
Figure 3.2: Artificial staleness in synchronous SGD compared to synchronous and asynchronous baselines, all with our usual learning rate for each model.

With four GPUs, baseline synchronous SGD has an effective batch size of 40 GB, compared to 10 GB in asynchronous. We fill in the two missing scenarios using synchronous SGD with a total effective batch size of 10 GB and asynchronous SGD with a batch size of 40 GB. Since GPU memory is limited, we simulate a larger batch size in asynchronous SGD by locally accumulating gradients in each processor four times before sending the summed gradient to the parameter server (Ott et al., 2018; Bogoychev et al., 2018).

Models with a batch size of 40GB achieve better BLEU *per update* when compared to its 10GB variant, as shown in Figure 3.1. However, synchronous SGD training still outperforms asynchronous SGD training—even with smaller batch size. Based on this experiment, we conclude that batch size is not the primary driver of the poor performance of asynchronously trained Transformers; however, it does have some lingering impact on final model quality. For RNNs, batch size and the distributed training algorithm had little impact beyond the early stages of training, continuing the theme that Transformers are more sensitive to noisy gradients.

3.4.3 Gradient Staleness

To study the impact of gradient staleness, we introduce staleness into synchronous SGD. Workers only pull the latest parameter once every U updates, yielding an average



(a) Transformer model with lr = 0.0006 (b) RNN model with lr = 0.0012

Figure 3.3: Artificial staleness in synchronous SGD with doubled learning rates. Transformers with a learning rate of 0.0006 and a staleness of 3 (synchronous and asynchronous) did not exceed 0.

staleness of $\frac{(U-1)}{2}$. Since asynchronous SGD has an average staleness of 3 with $N = 4$ GPUs, we set $U = 7$ to achieve the same average staleness of 3. Additionally, we attempted a lower average staleness of 2 by setting $U = 5$. We also observe the effect of doubling the learning rate so that the parameter moves twice as far, thereby introducing staleness in terms of model distance.

To focus on the impact of the staleness, we set the batch size to 40 GB total RAM consumption, be they 4 GPUs with 10 GB each in synchronous SGD or emulated 40 GB batches on each GPU in asynchronous SGD.

The results are presented in Figure 3.2. Staleness 3 substantially degrades Transformer convergence and final quality (Figure 3.2a). However, the impact of staleness 2 is relatively minor. We also continue to observe that Transformers are more sensitive than RNNs to training conditions.

The Transformer results worsen when we double the learning rate (Figure 3.3). With staleness 3, the model remained at 0 BLEU for both synchronous and asynchronous SGD, which is consistent with our earlier result (Table 3.1).

We conclude that staleness is primarily—but not wholly—responsible for the poor performance of asynchronous SGD in training Transformers. However, asynchronous SGD still under-performs in comparison to synchronous SGD with an artificial stale-

ness of 3 and the same batch size (40 GB). Our synchronous SGD training has consistent parameters across processors, whereas processors might have different parameters in asynchronous training. The staleness distribution might also play a role since staleness in asynchronous SGD follows a normal distribution (Zhang et al., 2016) while our synthetic staleness in synchronous SGD follows a uniform distribution.

3.5 Asynchronous Transformer Training

3.5.1 Accumulated Asynchronous SGD

Previous experiments have demonstrated that increasing the batch size and reducing staleness improves the final quality of asynchronous training. Increasing the batch size can be achieved by accumulating gradients before updating. We experiment with variations in three ways of accumulating gradients:

Local Accumulation: Gradients can be accumulated locally in each processor before sending it to the parameter server (Ott et al., 2018; Bogoychev et al., 2018). This approach scales the effective batch size and reduces communication costs since the workers communicate less often. However, this approach does not reduce staleness because the parameter server updates immediately after receiving a gradient. Therefore, this approach can be considered a vanilla asynchronous SGD with a larger batch size. We experiment with accumulating four gradients locally, resulting in a 40 GB effective batch size.

Global Accumulation: Each processor sends the computed gradient to the parameter server normally. However, the parameter server holds the gradient and only updates the model after it receives multiple gradients (Dean et al., 2012; Lian et al., 2015). This approach scales the effective batch size by N, assuming that we accumulate N gradients. This behaviour is similar to synchronous SGD where the update is scaled with more workers. Moreover, it has less staleness compared to vanilla asynchronous SGD since the parameter server updates less often. More precisely, the parameter server only updates once every N gradient pushes. Therefore, staleness is scaled down by a factor of N. However, it does not reduce communication costs since each processor communicates with the server as often as vanilla asynchronous SGD. We experiment with accumulating four gradients globally, resulting in a 40 GB effective batch size and 0.75 average staleness.

Combined Accumulation: Local and global accumulation can be combined to

Transformer

Comm.	accumulation		batch	avg.	speed	best	hours to X BLEU		
	local	global	size	staleness	(wps)	BLEU	33	34	35
synchronous	1	4	40 GB	0	36029	35.66	5.3	7.6	15.6
asynchronous	1	1	10 GB	3	39883	30.72	-	-	-
asynchronous	4	1	40 GB	3	45177	30.98	-	-	-
asynchronous	2	2	40 GB	1.5	43115	35.68	4.9	6.8	15.4
asynchronous	1	4	40 GB	0.75	39514	35.84	4.6	6.7	11.4

RNN

Comm.	accumulation		batch	avg.	speed	best	hours to X BLEU		
	local	global	size	staleness	(wps)	BLEU	32	33	34
synchronous	1	4	40 GB	0	23054	34.30	3.6	6.2	18.8
asynchronous	1	1	10 GB	3	24683	33.76	2.7	5.1	-
asynchronous	4	1	40 GB	3	27090	33.83	4.1	6.1	-
asynchronous	2	2	40 GB	1.5	25578	34.20	3.2	5.9	13.7
asynchronous	1	4	40 GB	0.75	24312	34.48	3.1	5.4	14.5

Table 3.3: Quality and convergence of asynchronous SGD with accumulated gradients on an English-to-German dataset. Dashes indicate that the model never reached the target BLEU.

gain the benefits of both reduced communication cost and reduced average staleness. In this approach, gradients are accumulated locally in each processor before being sent. The parameter server also waits and accumulates gradients before running an optimiser. We accumulate two gradients both locally and globally to yield a 40 GB effective batch size and 1.5 average staleness.

We tested the three gradient accumulation flavours on the English-to-German task with both Transformer and RNN models. Synchronous SGD also appears as a baseline. To compare results, we report the best BLEU, raw training speed and time required to reach several BLEU checkpoints. The results are presented in Table 3.3.

Asynchronous SGD with global accumulation improves the final quality of the model over synchronous SGD, albeit not meaningfully. This one change, accumulating every four gradients (the number of GPUs), restores quality in asynchronous methods. It also achieves the fastest time to reach near-convergence BLEU in both Transformer

Model	EN → DE		EN → FI		EN → RU	
newstest	2016	2017	2017	2018	2015	2018
Trans. + synchronous	35.66	28.81	18.47	14.03	29.31	25.49
Trans. + asynchronous	30.72	24.68	11.63	8.73	21.12	17.78
Trans. + asynchronous + 4x GA	35.84	28.66	18.47	13.78	29.12	25.25
RNN + synchronous	34.30	27.43	16.94	12.75	26.96	23.11
RNN + asynchronous	33.76	26.84	14.94	10.96	26.39	22.48
RNN + asynchronous + 4x GA	34.48	27.56	17.05	12.76	27.15	23.41

Table 3.4: The effect of global accumulation (GA) on translation quality for different language pairs in the development and test set, as measured using BLEU scores.

and RNN.

Although using local accumulation provides even faster raw speed, the model produces the worst quality among the other accumulation techniques. Asynchronous SGD with 4x local accumulation is essentially ordinary asynchronous SGD with a 4x larger batch size and a 4x lower update frequency. In particular, gradient staleness remains the same and does not improve the convergence per update. The performance of combined accumulation somewhat in the middle since it does not converge as rapidly as asynchronous SGD with full global accumulation, but not as poorly as asynchronous SGD with full local accumulation. Its speed is also in between, reflecting the communication costs. On the other hand, the RNN model is less sensitive to stale gradients. Hence, we can accumulate some of the gradients locally for improved speed without sacrificing quality.

3.5.2 Generalisation Across Learning Rates

Earlier in Table 3.1, we presented that asynchronous Transformer learning is very sensitive towards the learning rate. In this experiment, we use an asynchronous SGD with global gradient accumulation to train English-to-German translation at different learning rates. We then compare our result with vanilla synchronous and vanilla asynchronous SGD.

Our finding empirically demonstrates that asynchronous Transformer training while globally accumulating gradients is significantly more robust. As shown in Table 3.5, the model is now capable of learning at a higher learning rate while yielding comparable results to its synchronous variant.

	Communication		
	Sync.	Async.	Async
Learning Rate			+ 4x GA
0.0003	35.66	30.72	35.84
0.0006	35.42	0.00	35.81
0.0012	33.96	0.00	33.62
0.0024	29.35	0.00	1.20

Table 3.5: The performance of the asynchronous Transformer on English-to-German translation with 4x Global accumulations (GA) across different learning rates on the development set, as measured using BLEU scores.

3.5.3 Generalisation Across Languages

To test whether our findings on English-to-German can be generalised, we train two more translation systems using globally accumulated gradients. Specifically, we train English-to-Finnish (EN $\rightarrow$ FI) and English-to-Russian (EN $\rightarrow$ RU) models for the WMT 2018 task (Bojar et al., 2018). We validate our model on newstest2015 for EN $\rightarrow$ FI and newstest2017 for EN $\rightarrow$ RU. Then, we test our model on newstest2017 for EN $\rightarrow$ DE and newstest2018 for both EN $\rightarrow$ FI and EN $\rightarrow$ RU. The same network structures and hyperparameters are used as before.

The results presented in Table 3.4 empirically confirm that accumulating the gradient to obtain a larger batch size and a lower staleness in Transformer massively improves the result when compared to basic asynchronous SGD (+6 BLEU on average). The improvement is smaller in RNN experiment, but still substantial (+1 BLEU on average). We also have further confirmation that training a Transformer model with normal asynchronous SGD is impractical.

3.6 Related Work

3.6.1 Gradient Summing

Several papers wait and sum P gradients from different workers as a method of reducing staleness. In Chen et al. (2016), gradients are accumulated from different processors, and other processors cancel their process and restart from the beginning whenever the P gradients have been pushed. This is relatively wasteful since some computation

is thrown out and $P-1$ processors remain idle for synchronisation. Gupta et al. (2016) suggest that while restarting is not necessary, processors continue to idle while waiting for P to finish. Our proposed method follows Lian et al. (2015), in which an update occurs every time P gradients have arrived, while processors continually generate gradients without synchronisation.

Aside from gradient summing, an alternative direction to overcome a stale gradient is to reduce its effect on the model update. McMahan and Streeter (2014) dynamically adjusted the learning rate depending on the staleness. Moreover, Dutta et al. (2018) suggests completely ignoring stale gradient pushes.

3.6.2 Training with Noisy Gradients

In the opposite direction, some work has intentionally added noise to gradients or increased staleness, typically to cut computational costs. Dean et al. (2012) mention that communication overload can be reduced by reducing gradient pushes and parameter synchronisation frequency. In McMahan et al. (2017), each processor independently updates its local model and periodically synchronises the parameter by averaging across other processors. Furthermore, Ott et al. (2018) accumulates gradients locally before sending it to the parameter server. Bogoychev et al. (2018) also locally accumulates the gradient, but updates local parameters in between.

Lossy gradient compression via bit quantisation (Seide et al., 2014; Alistarh et al., 2017) or threshold-based sparsification are discussed in Chapter 4, which also introduces noisy gradient updates. Furthermore, these techniques store unsent gradients to be added into the next gradient, thus becoming stale. We later determine that, consistent with the results presented in this chapter, RNN is more robust towards compressed gradients, while Transformer-based models break completely. We reduce the compression noise to resolve this issue in Chapter 5.

3.7 Conclusion

We evaluated the behaviour of Transformer and RNN models under asynchronous training and divide our analysis based on two main different aspects of asynchronous training: batch size and stale gradient. Our experimental results indicate that:

- In general, asynchronous training damages the final BLEU of the NMT model. However, we found that the damage to Transformer is significantly more severe.

Also, asynchronous training requires a smaller learning rate to perform well.

- With the same number of processors, asynchronous SGD has a smaller effective batch size. We empirically show that training under a larger batch size setting can slightly improve the training convergence. However, the improvement is very minimal. The result from the asynchronous Transformer model is sub-par, even with larger batch size.

- Stale gradients serve a larger role in the training performance of an asynchronous Transformer. We have demonstrated that the Transformer models performed poorly by adding a synthetic stale gradient.

Based on the findings of our study, we suggest applying a modification in asynchronous training by accumulating a few gradients (e.g. the number of processors) in the server before applying an update. This approach increases the batch size while reducing the average staleness. We empirically show that this approach combines the high-quality training of synchronous SGD and high training speed of asynchronous SGD.

Chapter 4

Sparse Gradient Communication

This chapter addresses gradient compression to reduce communication cost between workers in parallel training. This chapter is based on Aji and Heafield (2017).

4.1 Introduction

Distributed training is essential for large neural networks on large data sets (Raina et al., 2009). We focus on data parallelism, in which nodes jointly optimise the same model on different parts of the training data. The main performance issue in data parallelism is the cost of communicating gradients and model updates between nodes. This is problematic because gradients have the same size as the model.

We find that gradient updates have a positive skewness coefficient(Zwillinger and Kokoska, 1999), with most being close to zero. Strom (2015) proposed a method to compress the communication by dropping gradients that are below a constant threshold. Dryden et al. (2016) improved this by using a ratio instead of a constant threshold. However, Dryden et al. (2016) tested the method on a toy MNIST task. This chapter re-investigate this approach on the actual NMT problem.

We focus on scaling NMT (Ñeco and Forcada, 1996; Bahdanau et al., 2014) and compare our findings with prior work on MNIST. NMT parameters are dominated by three large embedding matrices: source language input, target language input and target language output. While these matrices deal with vocabulary words, any mini-batch will only see a small fraction of the vocabulary, which makes the gradient updates more skewed compared to MNIST. Additionally, the NMT system consists of multiple parameters with different scales and sizes compared to MNIST's shallow three-layer network with uniform size.

Our idea is inspired by the skew of the gradients towards zero. The gradient for unseen words is zero in the input matrices and small in the output matrix due to normalising to form a probability distribution. Empirically, we find that even internal non-vocabulary matrices have skewed gradients, as shown in Figure 4.1. More formally, the gradients have a positive skewness coefficient (Zwillinger and Kokoska, 1999). The output embedding matrix is very skewed (65.2 skewness coefficient). All other internal matrices are positively skewed with an average skewness coefficient of 3.1.

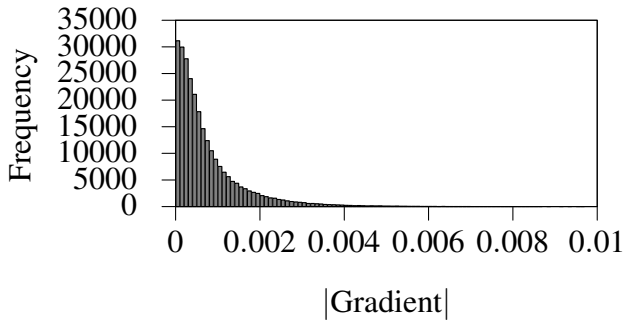


Figure 4.1: Histogram of gradient absolute values from an internal matrix, namely W_0 of the target-side RNN, taken from the system described in Section 4.4.

Given the near-sparsity of gradients, we map the smallest values to zero and send the largest values as a sparse matrix. Small values are accumulated locally so that they can accrue into larger changes.

4.2 Related Work

An orthogonal line of work optimises the SGD algorithm and communication pattern. Zinkevich et al. (2010) proposed an asynchronous architecture where each node can push and pull the model independently to avoid waiting for the slower node. Moreover, Chilimbi et al. (2014) and Recht et al. (2011) suggest updating the model without a lock to allow race conditions. Additionally, Dean et al. (2012) ran multiple mini-batches before exchanging updates to reduce the communication cost. Our work is a more continuous version in which the most important updates are sent between mini-batches.

The idea of compressing the gradient update is not new. Notably, 1-bit SGD (Seide et al., 2014) and Quantisation SGD (Alistarh et al., 2016) function by converting the

gradient update into a 1-bit matrix, thereby significantly reducing data communication. Strom (2015) proposed a threshold quantisation that only sends gradient updates that are larger than a pre-defined constant threshold. However, we must know the gradient scale before we can define a sensible threshold. Furthermore, since the gradient scale might change throughout the training, Dryden et al. (2016) proposed a method to recompute the threshold based on a given proportion.

4.3 Sparse Gradient Exchange

We sparsify gradient updates by choosing a threshold, and only sending gradients with an absulute value larger than the threshold, dubbing this Gradient Dropping. This approach is slightly different from Dryden et al. (2016) as we used a single threshold based on absolute value, instead of dropping per-individual row as well as sparsify the positive and negative gradients separately. We found out that our approach worked well and simpler to implement.

Small gradients can accumulate over time and we find that zeroing them damages convergence. Following Seide et al. (2014), we remember residuals (in our case dropped values) locally and add them to the next gradient.

Algorithm 1 Gradient dropping algorithm given gradient ∇ and dropping rate R.

function GRADDROP(∇, R)

$\nabla += residuals$

Select $threshold$: R% of $|\nabla|$ is smaller

$dropped \leftarrow 0$

$dropped[i] \leftarrow \nabla[i] \forall i : |\nabla[i]| > threshold$

$residuals \leftarrow \nabla - dropped$

return $sparse(dropped)$

end function

Gradient dropping is shown in Algorithm 1. This function is applied to all data transmissions, including parameter pulls encoded as deltas from the last version pulled by the client. To compute these deltas, we store the last pulled copy server-side. While we also store the last pulled copy per client, the server is responsible for $1/N$th of the parameters for N clients; therefore, memory is constant.

The selection to obtain the threshold is expensive (Alabi et al., 2012). However, this can be approximated. We sample 0.1% of the gradient and obtain the threshold by

running selection on the samples.

We can select a threshold locally to each matrix of parameters or globally for all parameters. In the experiments, we find that layer normalisation (Lei Ba et al., 2016) makes a global threshold work. Therefore, we use layer normalisation with one global threshold by default. Prior work does not address this possible issue.

4.4 Experiment

We experiment with an image classification task based on an MNIST dataset (LeCun et al., 1998) and Romanian→English neural machine translation system.

For our image classification experiment, we build a fully connected neural network with three 4069-neuron hidden layers. We use AdaGrad with an initial learning rate of 0.005 and a mini-batch size of 40. This setup is identical to the experiment by Dryden et al. (2016).

Our NMT experiment is based on the Marian implementation of Sennrich et al. (2016a), which won first place in the 2016 Workshop on Machine Translation [1]. It is based on an attentional encoder-decoder GRU with 119M parameters and a default batch size of 80. We save and validate every 10000 steps and select four saved models with the highest validation BLEU, and then average them into the final model. AmuNMT (Junczys-Dowmunt et al., 2016) is used for decoding with a beam size of 12. Our test system has PCI Express 3.0 x16 for each of 4 NVIDIA Pascal Titan Xs. The following experiments use asynchronous SGD, although our method also applies to synchronous SGD.

4.4.1 Drop Ratio

Dryden et al. (2016) sparsify the gradient to $1/32$ of its original size. However, based on our findings on the gradient skewness, we suggest that this ratio can be increased for further compression. To find an appropriate dropping ratio $R\%$, we attempted 90%, 99%, and 99.9%, then measured performance in terms of loss and classification accuracy or translation quality approximated by BLEU (Papineni et al., 2002b) for image classification and NMT tasks, respectively. We used a global threshold.

Figure 4.2 shows that the model still learns after dropping 99.9% of the gradients, although it becomes very unstable. It also damages the BLEU score by 1.5 points over

[1] https://github.com/marian-nmt/marian-examples/tree/master/training-basics

the baseline. The model converged slightly slower by dropping 99% of the gradients, though it can catch up to a comparable BLEU score afterwards, despite exchanging 50x less data with offset-value encoding.

A similar pattern can be observed in the MNIST experiment. However, MNIST is easier to train since the models reached high accuracy in the early stage of training. A 99.9% drop rate is shown to be more stable in MNIST and can reach to respectable accuracy (only losing 0.002% accuracy compared to the baseline), though it requires more updates to reach this point. We suggest that gradient can be dropped more aggressively on simpler tasks.

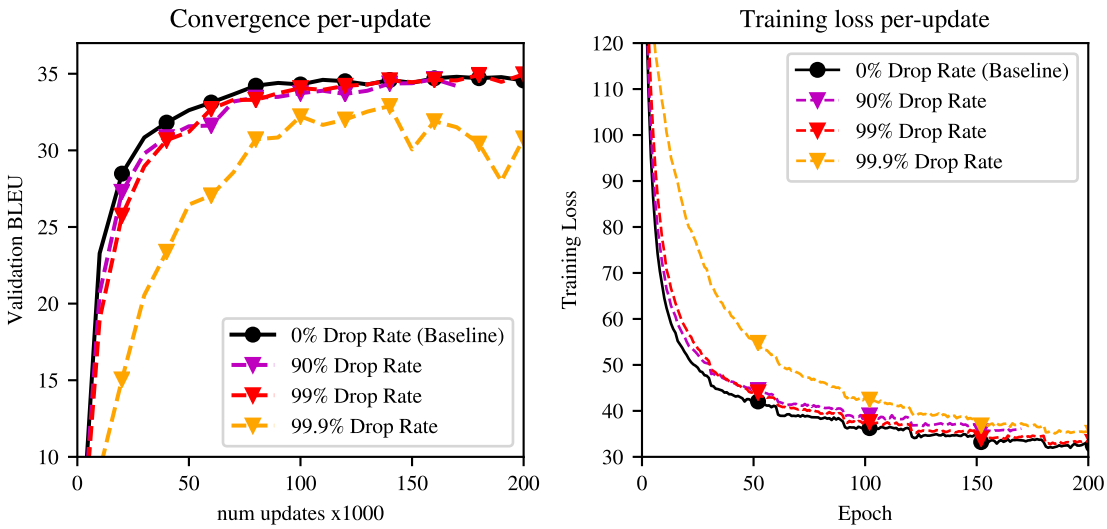


Figure 4.2: NMT: Training loss and validation BLEU for different dropping ratios.

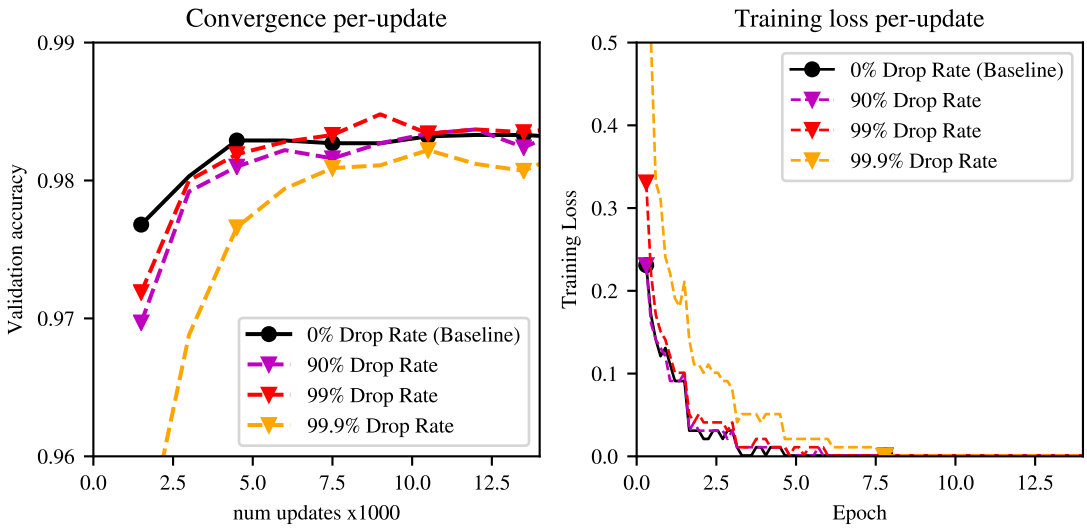


Figure 4.3: MNIST: Training loss and validation BLEU for different dropping ratios.

4.4.2 Local vs Global Threshold

Since parameters may not be on a comparable scale, we experiment with local thresholds for each matrix or a global threshold for all gradients so, as discussed in Section 4.3. We also investigate the effect of layer normalisation. We use a drop ratio of 99%, as previously suggested.

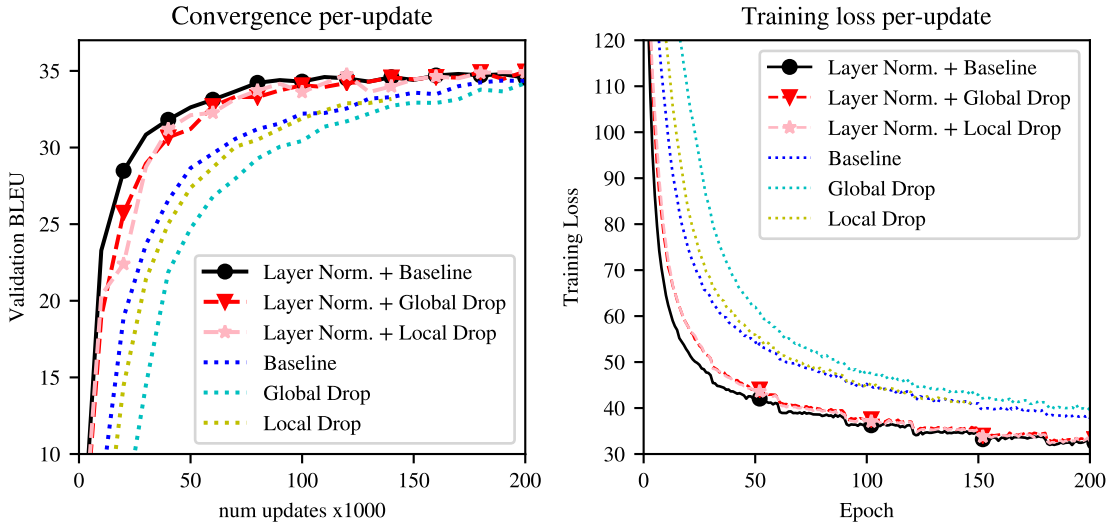


Figure 4.4: NMT: Comparison of local and global thresholds with and without layer normalization.

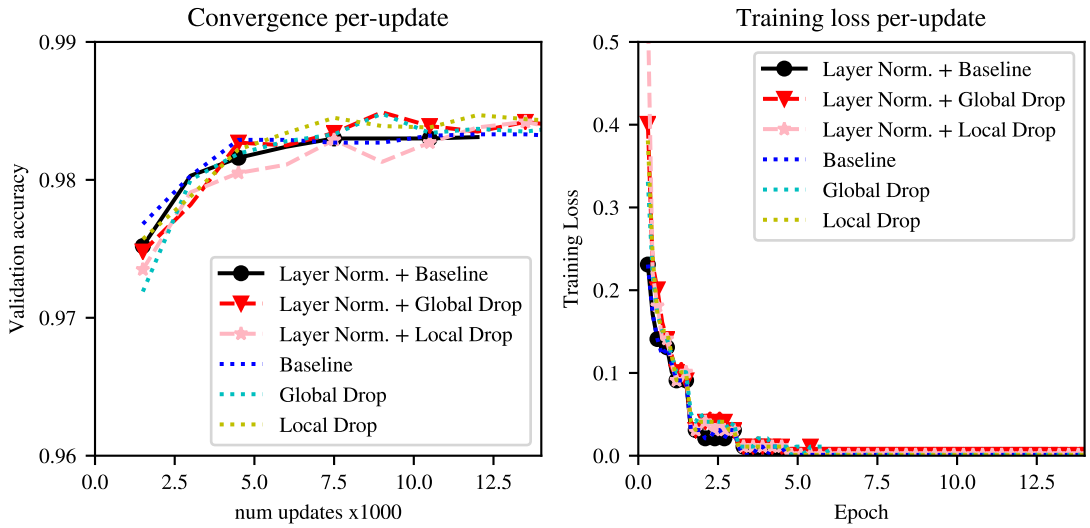


Figure 4.5: MNIST: Comparison of local and global thresholds with and without layer normalization.

The results show that layer normalisation has no visible impact on the MNIST experiment. Similarly, there is no significant difference between dropping the gradient

Drop Ratio	words/sec (NMT)	image/sec (MNIST)
0%	13100	2489
90%	14443	3174
99%	14740	3726
99.9%	14786	3921

Table 4.1: Training speed with various drop ratios.

locally or globally. On the other side, our baseline NMT system converged poorly without layer normalisation. Without layer normalisation, parameters are on various scales and global thresholding performed the worst. With layer normalisation, both global and local thresholding performed similarly.

4.4.3 Speed Benchmark

Raw Speed Measurement

Gradient dropping cuts communication cost significantly, thus improving raw speed in terms of words/image processed per second, as shown in Table 4.1. The speed improvement of dropping 99.9% of gradients is negligible compared to 99%. Based on the trade-off between a minimal speed gain with a significant quality reduction as shown in Section 4.4.1, we suggest that communicating only 1% of the gradient is efficient enough. We further demonstrate this by measuring the time spent on inter-GPU communication in Figure 4.6. Since batch size determines the ratio between communication and computation, we test a range of batch sizes.

Figure 4.6 divides the total time into three categories. Communication time is the time to transfer data between nodes, including wait time due to synchronisation. Computation time is the time to complete the forward and backward pass and apply updates with the optimiser. Lastly, "dropping" indicates compression overhead, including finding thresholds and sparse encoding.

As shown in Figure 4.6, gradient dropping substantially reduces communication cost. Reducing communication indirectly reduces the computation cost since there is less overlap between communication and computation. Additionally, communication cost is constant across different batch sizes, resulting in the speed ratio being higher with lower batch sizes (Table 4.2). Unfortunately, the communication cost in this ex-

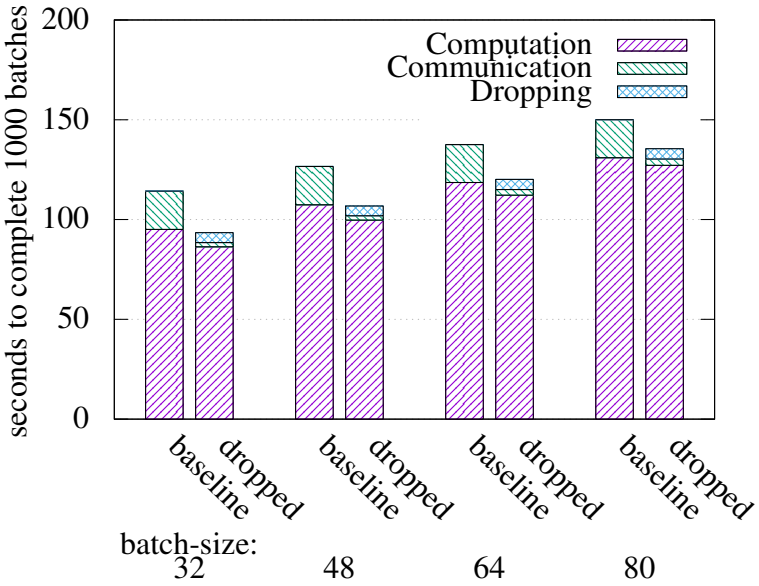


Figure 4.6: Breakdown of training time for various batch sizes.

periment is less than 20% of the total time spent. Therefore, the relative improvement is small.

Batch Size	Base (w/s)	Drop (w/s)	Improvement
32	6989	8553	1.22x
48	9442	11205	1.19x
64	11613	13304	1.15x
80	13317	14740	1.11x

Table 4.2: Speed, in words per second, for various batch sizes.

Convergence Rate

Ultimately, training speed indicates how fast the model can converge. We measure this by the time required to reach a certain quality threshold. In the MNIST experiment, we train the model for 20 epochs, as per (Dryden et al., 2016). In an NMT experiment, we tested this with batch sizes of 80 and 32 and trained for 13.5 hours.

While gradient dropping improves the raw speed, it also slightly damages the convergence per update. While we process each batch faster with gradient dropping, the model requires more batches to reach the same quality. These cancel each other out, ultimately yielding no improvement in terms of accuracy or BLEU score over time, as

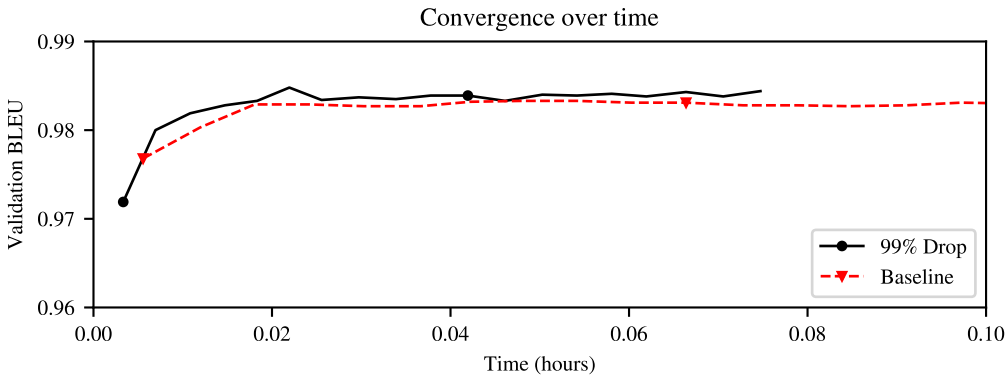


Figure 4.7: MNIST classification accuracy over time.

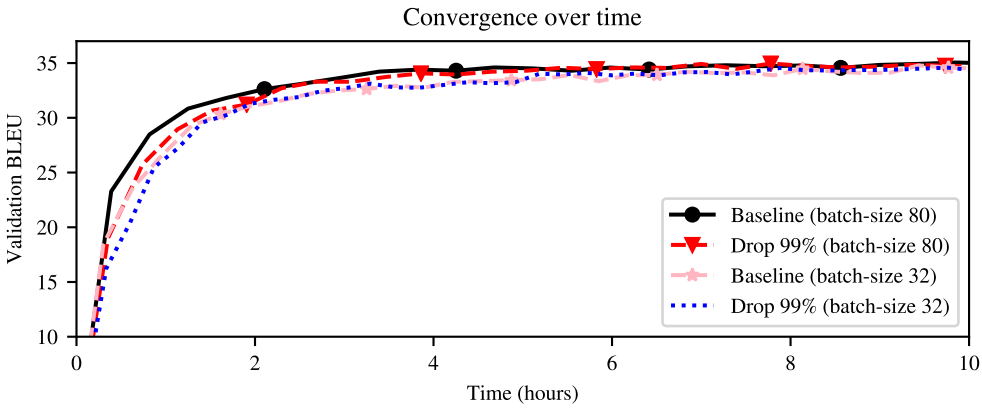


Figure 4.8: NMT validation BLEU and loss over time.

Method	**Test BLEU**	**Time to reach 33 Dev. BLEU**
batch-size 80		
+ baseline	34.51	2.6 hours
+ 99% grad-drop	34.40	2.7 hours
batch-size 32		
+ baseline	34.16	4.2 hours
+ 99% grad-drop	34.08	3.2 hours

Table 4.3: Summary of BLEU score obtained.

shown in Figure 4.7 and Figure 4.8.

To better investigate the convergence, we capture the time required for the model to reach a near-convergence level of 33 BLEU, as shown in Table 4.3. Notably, final

BLEU scores are essentially unchanged. Our algorithm converges 23% faster than the baseline when using the sub-optimal batch size of 32. However, the model trained under this batch size under-performs. Unfortunately, we do not observe any gain with a batch size of 80, which is a setting with rapid communication (15.75 GB/s theoretical over PCI express 3.0 x16). This leads us to hypothesise that gradient dropping will be useful in multi-node scenarios, where communication is far more expensive.

4.4.4 One-bit Quantisation

We compare our results with the 1-bit quantisation technique Seide et al. (2014). This quantisation is column-wise, where each gradient is replaced by their positive or negative column mean. We can also obtain further compression by stacking 1-bit quantisation after dropping the gradient. We apply the quantisation after gradient dropping with a 99% drop rate, layer normalisation and a global threshold.

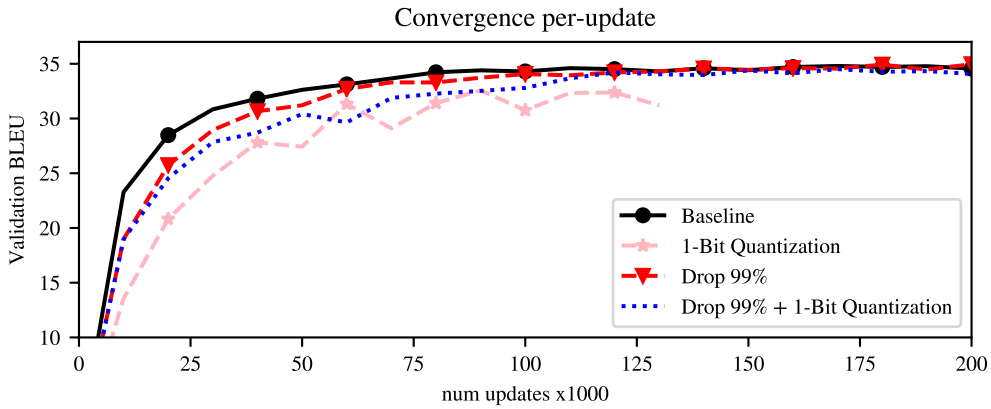


Figure 4.9: BLEU score over time using 1-bit Quantisation method.

The result in Figure 4.9 shows that 1-bit quantisation slows down the convergence more than gradient dropping. Furthermore, the training is unstable and converged to a lower quality. However, the model managed to converge better if we stack both together. Since most of the gradients are near zero, their mean value would be small; therefore, 1-bit quantisation might generate more quantisation error. Since we only send large gradients with gradient dropping, it results in a larger mean for the quantisation.

The 1-bit quantisation resulted in a 32x smaller communication cost. Stacking it with our gradient dropping using a 99% drop rate provides 100x reduced communication cost since only 1 bit must be sent for the sign and 31 bits for the index. However,

since communication cost is already negligible after dropping 99% of the gradients, further stacking it with 1-bit quantisation does not improve the raw speed. With more damage to the convergence, we suggest that 1-bit quantisation may not be compatible with every scenario.

4.5 Conclusion

Gradient updates are positively skewed, with most being close to zero. This can be exploited by delaying 99% of gradient updates locally, thereby reducing communication size by 50x with coordinate-value encoding. The dropping threshold can be computed globally or locally for each layer. However, our NMT system consists of many parameters with different scales; therefore, layer normalisation is necessary for global thresholding. On the other hand, MNIST seems to work with any configurations we tried.

Prior work suggested that 1-bit quantisation can be applied to compress communication. However, we empirically determined that this is not true for NMT, which we attribute to skew in the gradients. However, stacking with sparsification is likely to be sufficient since it separates large movers from small changes.

While the model trained with sparse gradients achieves comparable quality, the speed improvement is insignificant. Our experiment with 4 Titan Xs shows that, on average, only 17% of the time is spent communicating (with a batch size of 32) and we achieve a 22% raw speed increase. Additionally, a compressed gradient is reduces the convergence per update, thus yielding no speed increase in terms of the time required to reach a certain BLEU score. Our next experiment involves testing this approach on systems with expensive communication costs (e.g., multi-node environments). We also experiment with restoring the sparse gradient quality to improve the convergence per update.

Chapter 5

Sparse Gradient with Local Context

In this chapter, we incorporate local gradients to restore the sparse gradient quality in gradient compression that was discussed in Chapter 4. This chapter is based on Aji et al. (2019).

5.1 Introduction

In recent years, neural network models have grown dramatically in terms of the number of parameters (Wen et al., 2017; Huang et al., 2019); therefore, exchanging gradients during data-parallel training is costly in terms of both bandwidth and time—especially in a distributed setting.

In Chapter 4, we discuss a solution to reduce communication cost by sending only the top 1% of the largest gradients in terms of absolute values. Related communication compression methods include synchronising less often (Konečnỳ et al., 2016; Ott et al., 2018; Bogoychev et al., 2018) and quantisation (Seide et al., 2014; Alistarh et al., 2016).

Since these compression methods are lossy, each node's locally-computed gradient is not immediately reflected in the global gradient. Our experiments in Chapter 4 showed that gradient compression damages the model's convergence. In this chapter, we show that sparse gradient breaks the Transformer model (Vaswani et al., 2017), which is known to be sensitive to noisy gradients (Chen et al., 2018; Ott et al., 2018). We aim to repair the compressed gradient by combining it with local gradients to improve the trade-off between convergence and compression rates.

In this chapter, we apply the gradient dropping method explored in Chapter 4 to reduce the inter-node communication during distributed neural network training, which

leads to faster training speed but reduced model convergence rate. We find that combining the sparse global gradient with the dense local gradient improves convergence. However, adding local information implies that node parameters will diverge over time. We address this by periodically averaging the model inspired by Konečnỳ et al. (2016) to achieve faster end-to-end training time.

5.2 Related Work

5.2.1 Sparse Gradient Compression

In Chapter 4, we proposed a gradient compression technique exploiting its skewness property by sending only a sparse matrix of large gradients. Unsent gradients are added to the next gradient before compression (Seide et al., 2014).

Algorithm 2 Sparse SGD on node n at timestep t

1: **procedure** SPARSESGD(L_t^n) ▷ L_t^n is local gradient of node n at step t.
2: $S_t^n \leftarrow GradDrop(L_t^n, threshold)$
3: $G_t \leftarrow AllReduce(S_t^n)$
4: $ApplyOptimizer(G_t)$
5: **end procedure**

An outline of the sparse gradient update is presented in Algorithm 2. We first compress the local gradient L_t^n with gradient dropping (the *GradDrop* function is defined in Algorithm 1 from the previous chapter) and return the sparsified gradient S_t^n, which will be used for the parameter update. Different from the previous work in Chapter 4, we use synchronous training. With synchronous training, parameter updates run redundantly in all nodes so that only gradients are sent over the network. Alternatively, aggregating the gradients with a sharded parameter server architecture similar to asynchronous training is an option. However, this method requires us to re-compress the pulled parameter's difference, which we consider slow as we have to run the compression twice. This approach is also potentially more harmful since we have more compression and stalled updating.

Notably, the sum of sparse gradients is less sparse. We can send the summed gradient as it is (Lin et al., 2018) or again take the top 1% of summed gradients. Similar to the issue in the parameter server, we found that re-compressing the gradient is slower than sending less sparse gradients. In a case where we need more compression, we

suggest tuning the initial compression rate instead.

5.2.2 Federated Averaging

Another way to reduce the bandwidth cost in multi-node training is by reducing the communication frequency (Konečnỳ et al., 2016). In federated averaging, workers do not exchange gradients. Instead, each worker uses its local gradient to update its local parameters. Each worker updates their local parameters by averaging across other nodes once every few steps.

$$\begin{aligned} \theta_t^n &= \theta_t^n - L_t^n \\ \theta_t^n &= \frac{\sum_{i=1}^{N} \theta_t^i}{N} \quad \text{if } t \bmod P \text{ is } 0 \end{aligned} \tag{5.1}$$

Formally, let θ_{it} be the i-th node's parameter at the time step t. The parameter is updated with the local gradient L_t^n. Once per-P steps, however, the parameter is then averaged across other workers.

In contrast to gradient dropping, federated averaging primarily uses the workers' local gradients for parameter updates. Gradients from other workers are not directly communicated.

5.3 Combining With Local Gradients

Recent work suggests that the Transformer is sensitive to noisy gradients, thus resulting in substantially worse models (Chen et al., 2018; Ott et al., 2018). We also confirm the Transformer's sensitivity in Chapter 3. Consistent with these findings, both gradient sparsification and federated averaging yield abysmally low-quality Transformer models in our experiments. In gradient sparsification, noise comes from both thresholding and the error feedback mechanism, resulting in stale gradients. Federated averaging also introduces stale updates since this approach delays model synchronisation. Previous work has shown that both noisy and stale gradients damage the model's quality (McMahan and Streeter, 2014; Ott et al., 2018; Dutta et al., 2018).

To address noisy updates in gradient sparsification, we combine the compressed global gradient and the uncompressed locally-computed gradient to better approximate the true global gradient. Formally, let G_t be the compressed global gradient at time t and L_t^n be the gradient computed locally on node n. These will be combined

Algorithm 3 Sparse SGD with local gradient incorporation on node n at timestep t

procedure SPARSESGD(L_t^n) ▷ L_t^n is local gradient of node n at step t.
  $S_t^n \leftarrow GradDrop(L_t^n, threshold)$
  $G_t \leftarrow AllReduce(S_t^n)$
  $C_t^n \leftarrow G_t - S_t^n + L_t^n$ ▷ Incorporate local context
  $ApplyOptimizer(C_t^n)$
  if t % $sync\_period = 0$ **then**
    $SynchronizeParams()$ ▷ Synchronise parameters across nodes
  end if
end procedure

into C_t^n, which will be used to update the parameters. Since the local gradients are different between nodes, the parameters will diverge. Therefore, we also consider re-synchronising the parameters every so often. The sparse gradient updates with local gradient incorporation and periodic parameter synchronisation are outlined in Algorithm 3.

5.3.1 Incorporating Local Gradients

An arguably naïve method sums the two gradients. With the scale-invariant Adam optimiser, summing is equivalent to averaging.

$$C_t^n = G_t + L_t^n$$

However, some of the locally-computed gradients were sent out and became part of the global gradient, so they will be double-counted by the sum. To compensate, we can subtract the gradients S_t^n sent by node n.

$$C_t^n = G_t - S_t^n + L_t^n$$

The term $G_t - S_t^n$ equals to the sum of all sparse gradients from other nodes (or approximates it when the all-reduce function compresses the result). The local gradient L_t^n is used for updating and does not include the error feedback term E_t^n to prevent applying gradients multiple times while they are pending in error feedback.

5.3.2 Periodic Synchronisation

Nodes will diverge because local gradients differ. Therefore, models are averaged periodically. We average parameters (Konečnỳ et al., 2016) every 500 steps with a minimal impact on speed. The sparse gradient updates with local gradient incorporation and periodic parameter synchronisation are outlined in Algorithm 3.

In the limit, a gradient is applied twice. First, directly from a local update eventually reaches the other nodes via periodic averaging. Second, it accumulates with other gradients as compressed gradient and applied as part of a global update.

5.4 Experimental Setup

We use Marian (Junczys-Dowmunt et al., 2018) to train on nodes with 4xP100s. Multi-node experiments use four of these nodes, each connected with 40Gb Mellanox Infiniband. These scenarios will be abbreviated as 1x4 (one node with four GPUs) and 4x4 (four nodes with four GPUs each).

5.4.1 Model and Dataset

We perform our neural machine translation experiments on the following architectures.

Transformer: We train a Transformer model with six encoder and six decoder layers with tied embeddings. The model has 62M parameters. We train the model on the WMT 2017 English-to-German dataset with back-translated monolingual corpora (Sennrich et al., 2016b) and byte-pair encoding (Sennrich et al., 2016c) consisting of 19.1M sentence pairs. Model performance is validated on newstest2016 and tested on newstest2017.

Deep RNN: We also train a deep RNN model (Sennrich et al., 2017) with eight layers of bidirectional LSTM consisting of 225M parameters. We train the model with the same English-to-German dataset from the Transformer experiment.

Shallow RNN: Our shallow RNN model is based on the system by Sennrich et al. (2016a) and is a single layer bidirectional encoder-decoder LSTM with attention consisting of 119M parameters. We train this model on the WMT 2016 Romanian-to-English dataset consisting of 2.5M sentence pairs. We also apply byte-pair encoding to this dataset. Model performance is validated on newsdev2016 and tested on newstest2016.

We apply layer normalisation (Lei Ba et al., 2016) and exponential smoothing to train the model for eight epochs of training.

5.4.2 Scaling Hyperparameters

When scaling the workers by a factor of N, we expect to see:

- N times larger batch size, therefore;
- N times fewer updates given the same amount of data.
- Assuming the communication cost between nodes are free, we expect the same number of update per given time.

If we scale the number of workers without adjusting the hyperparameters, we should expect mostly the same convergence curve per update as the baseline, with slight improvement. Practically, the convergence is slightly better with more workers due to more stable gradients, as reported in Chapter 3. Assuming an equal number of updates per time, we will not see any significant speed increase. In practice, considering the communication cost, multi-node training has fewer updates per time and thus converges slower.

We apply several adjustments to the hyperparameters to accommodate the larger effective batch size of multi-node synchronous SGD.

Learning rate: using the scale-invariant Adam optimiser, parameters move at the same magnitude regardless of the gradient size. This is inefficient since there are fewer updates within the same amount of data with a larger batch size; thus, the model moves N times less far. Therefore, we linearly scale the learning rate in all multi-node experiments, as suggested by Goyal et al. (2017). On one node, we use a learning rate of 0.0003 for Transformer and deep RNN models, and 0.001 for the shallow RNN model. These values are multiplied by 4 for the 4-node setting. The single-node learning rates were optimised such that further increasing them damages performance.

Warm-up: If we leave the warm-up rate unscaled, the model will reach the maximum learning rate slower. For example, single-node training reached 16k steps within a single epoch, while reaching four epochs (half-way through training) in multi-node training. To obtain the same learning rate given the same amount of data, we linearly scale down the learning rate warm-up period. We use the warm-up step of 16k and 4k for the Transformer and RNN experiments, respectively. These values are divided by

	Transformer		Deep RNN		Shallow RNN	
Model	dev	test	dev	test	dev	test
Multi-node (4x4)	35.39	28.78	34.45	27.81	35.45	34.45
4x4 + gradient dropping	0.00	0.00	34.38	27.50	35.20	33.89
4x4 + federated averaging	0.00	0.00	34.33	27.42	35.25	33.93
4x4 + grad. dropping + local update	35.07	28.50	34.52	27.68	35.35	34.45

Table 5.1: Training quality of multi-node training with gradient compression techniques, as measured by BLEU scores.

4 for the 4-node setting. Similarly, these values were optimised since lowering them damages quality.

In all of our experiments, we use a memory budget of 10GB per GPU to dynamically fit as many sentences as possible, corresponding to an average of 450 and 250 sentences per batch per GPU for Ro-En and En-De, respectively. We follow the remaining hyperparameter settings, as suggested in the papers (Vaswani et al., 2017; Sennrich et al., 2017, 2016a).

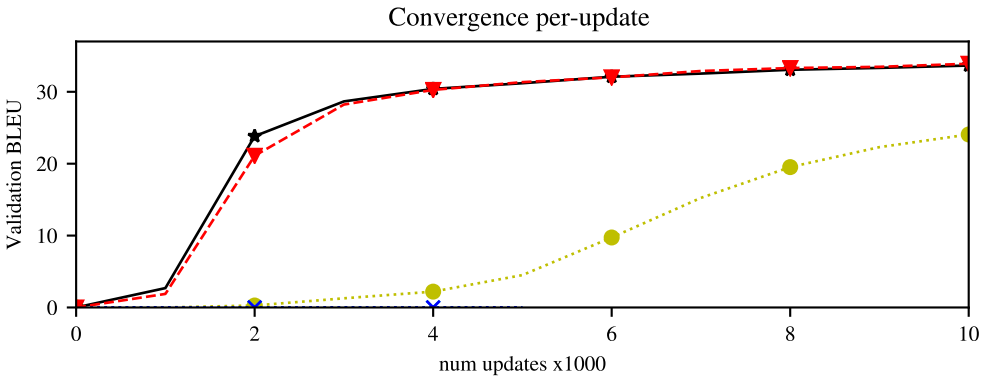
The raw words/second speed increase is only up to 2.7x faster from the expected 4x, signifying a communication bottleneck. With correct scaling, the model is also expected to reach a near-convergence level 2.7x faster.

5.5 Results and Analysis

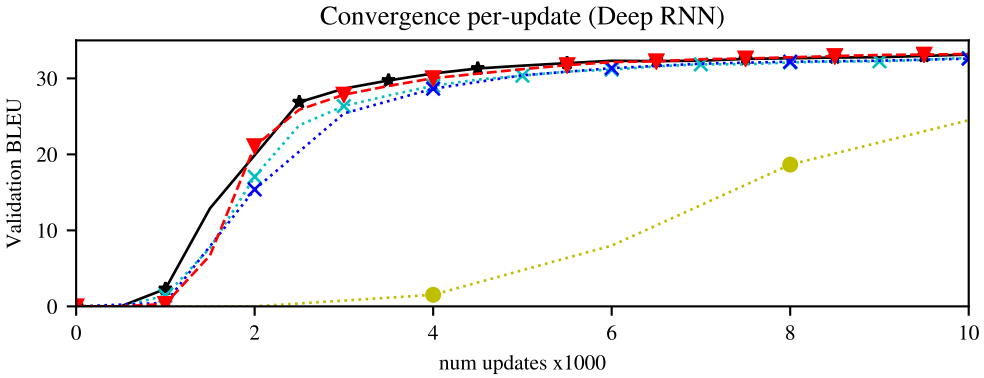
5.5.1 Restoring Quality

We approximate an impact on quality by measuring the BLEU score (Papineni et al., 2002b) obtained per update by experimenting with both RNN and Transformer systems. The baselines are vanilla synchronous SGD, gradient dropping (Chapter 4) and federated averaging (Konečnỳ et al., 2016). For gradient dropping, we perform a drop ratio warm-up, gradually increasing it to 99% after 1000 steps. We average the model every 20 steps in a federated averaging experiment and every 500 steps in our proposed method.

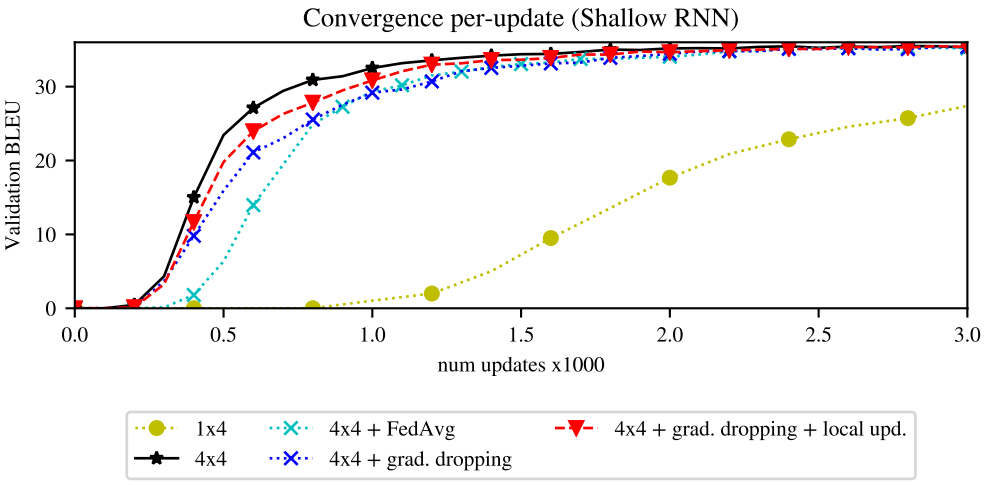
Figure 5.1 shows the BLEU score per update. Gradient dropping and federated averaging reduce gradient quality, resulting in the improvement per update becoming slower. In the Transformer case, the model is entirely incapable of training. Local gra-



(a) Transformer En-De



(b) Deep RNN En-De



(c) Shallow RNN Ro-En

Figure 5.1: Model convergence per update on gradient dropping with local gradient update. We focus on the early stage of the training before the BLEU scores converged. Training Transformer with gradient dropping or federated averaging yielded 0 BLEU.

Model	Words/	Raw speed-up	Time to	conv. speed-up
Transformer (En-De)	second	(1x4 / 4x4)	conv.	(1x4 / 4x4)
Single-node (1x4)	36029	-	7.61h	-
Multi-node (4x4)	95691	2.7x / -	3.52h	2.1x / -
Multi-node (12x4)	252709	7.5x / 2.8x	1.84h	4.1x / 1.9x
4x4 + grad. dropping + local	127516	3.7x / 1.4x	2.75h	2.7x / 1.3x
12x4 + grad. dropping + local	370673	10.2x / 3.8x	1.44h	5.2x / 2.4x
Deep RNN (En-De)				
Single-node (1x4)	18205	-	23.68h	-
Multi-node (4x4)	42930	2.4x / -	10.59h	2.2x / -
4x4 + grad. dropping	60090	3.3x / 1.4x	8.94h	2.6x / 1.2x
4x4 + federated averaging	66149	3.6x / 1.5x	9.50h	2.5x / 1.1x
4x4 + grad. dropping + local	59747	3.3x / 1.4x	6.80h	3.5x / 1.5x
Shallow RNN (Ro-En)				
Single-node (1x4)	36466	-	2.37h	-
Multi-node (4x4)	75641	2.1x / -	1.05h	2.3x / -
4x4 + grad. dropping	118189	3.2x / 1.6x	0.94h	2.5x / 1.1x
4x4 + federated averaging	124273	3.4x / 1.6x	1.06h	2.2x / 1.0x
4x4 + grad. dropping + local	117756	3.2x / 1.6x	0.85h	2.8x / 1.2x

Table 5.2: Speed performance of gradient dropping with local gradient update compared to several baselines. Time to convergence is the time required to reach 34.5 BLEU (Transformer & Shallow RNN) or 33.5 BLEU (Deep RNN).

dient incorporation improves the sparse gradient quality and improves convergence per epoch over gradient dropping. In all architectures, the model achieved a comparable training curve compared to the uncompressed multi-node training.

Table 5.1 summarises model performance in terms of BLEU. With local gradient incorporation, the models obtained a better final quality, performing closer to uncompressed multi-node training. Local gradient incorporation enables Transformer to train with a sparse gradient, albeit with slight quality degradation (0.28–0.32%). This result confirms Transformer's sensitivity to noisy updates and the ability of local gradients to mostly repair them.

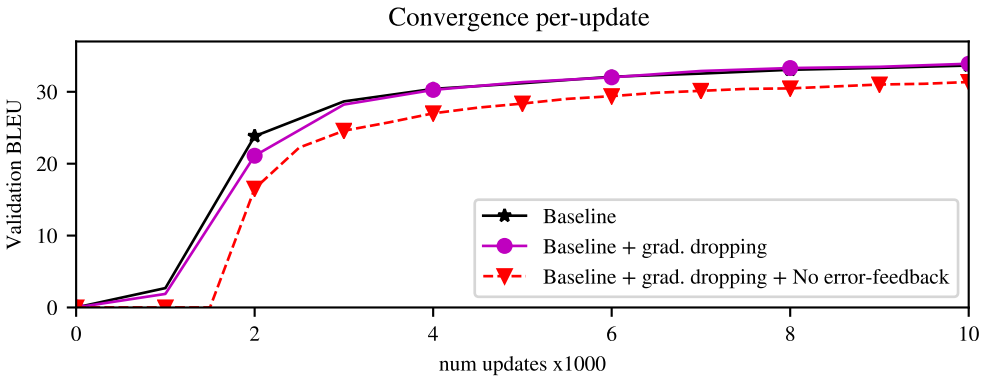


Figure 5.2: Convergence of Transformer model trained with sparse gradient without the error-feedback mechanism.

5.5.2 Removing Error Feedback Mechanism

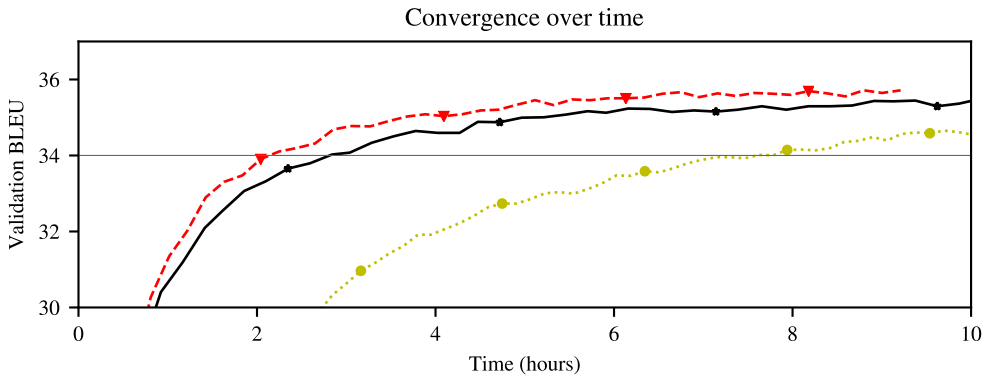
The error feedback mechanism was designed so that no gradients are removed, as we simply delay them. However, with local context incorporation, the gradient will be passed to the parameters, either as a local update or exchanged in a sparse gradient. Therefore, we explore whether removing the error feedback mechanism affects training performance.

Our experimental results on Transformer architecture indicate that the model is now capable of training without the error feedback mechanism if the local gradient is incorporated (Figure 5.2). Without the local context, the model diverged and reached 0 BLEU. However, it is evident that the training is rather slow and the final quality is damaged (-2 BLEU from the baseline). Therefore, error feedback remains necessary to maintain translation quality.

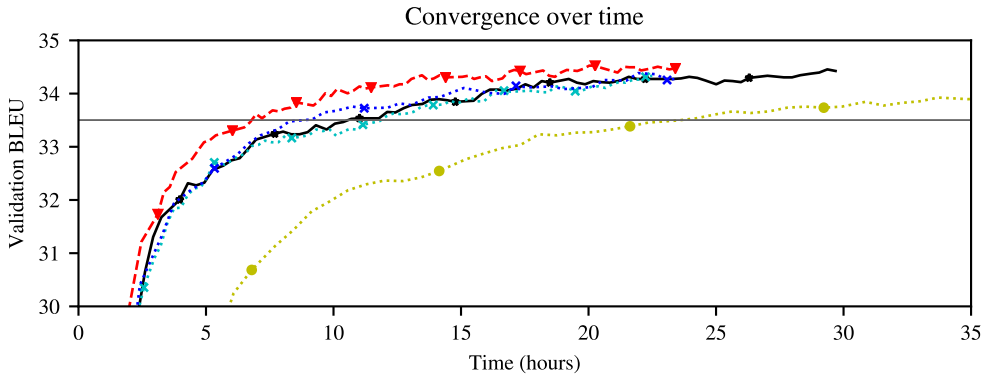
5.5.3 Improving Training Speed

We measure the speed improvement of our proposed method by capturing the raw processing speed and time to reach a certain BLEU. We compare it to both gradient dropping and federated averaging. We also measure the training efficiency by comparing the results with a single-node system. For the Transformer, we exclude vanilla gradient dropping and federated averaging as the models fail to train.

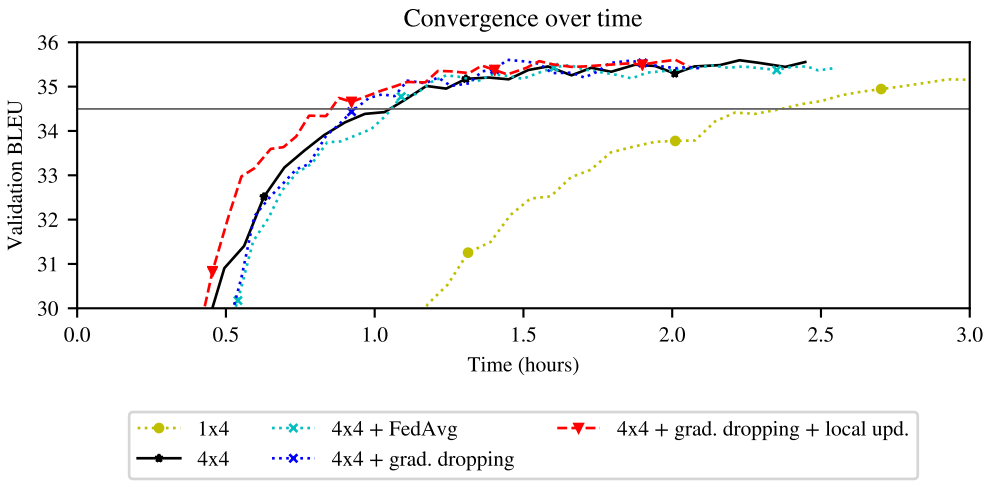
Table 5.2 summarises our experiments. Gradient dropping reduces network traffic and significantly improves raw training speed in the multi-node setting by up to 3.7x over the single-node setting, and up to 1.6x faster raw speed over the uncom-



(a) Transformer En-De



(b) Deep RNN En-De



(c) Shallow RNN Ro-En

Figure 5.3: Convergence over time of gradient dropping with local gradient update.

pressed multi-node setting. Federated averaging is faster since there is no additional communication overhead for each step and no extra computational cost for sparse gradient compression. Finally, our method incurs the combined cost of gradient dropping,

occasional federated averaging and local updates; therefore, it is slower than gradient dropping at raw speed but still substantially faster than uncompressed multi-node training.

While vanilla gradient dropping and federated averaging have better raw speed, there is no clear improvement in convergence speed as noisy gradients damage the convergence. Local gradient updates restore the gradient and improve the convergence speed. In our RNN experiments, the convergence speed increase is closer to the raw speed increase (up to 3.5x single-node performance).

Notably, the Transformer convergence rate increases more slowly than raw batch processing speed. While the rule of thumb is to scale learning rate linearly with batch size (Goyal et al., 2017), we showed in Chapter 3 that the Transformer model is also sensitive to high learning rates. We obtained a 2.1x convergence speed increase from the 2.7x raw speed increase. In contrast, raw and convergence speed increases in RNN models are comparable.

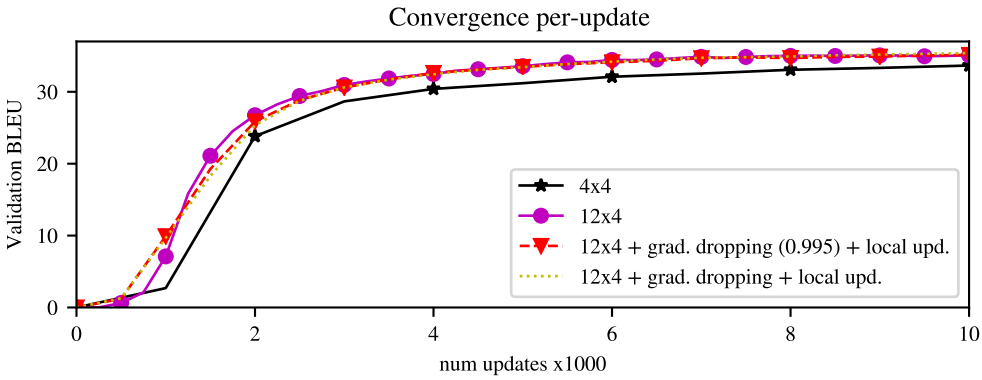
Compression results are dependent on the ratio between computation and network bandwidth in a system, as well as model size. Since the method reduces network load, we would expect to see even larger speed improvement with commodity hardware instead of the 40Gb Infiniband network used in our experiments.

5.5.4 Large-scale Experiment

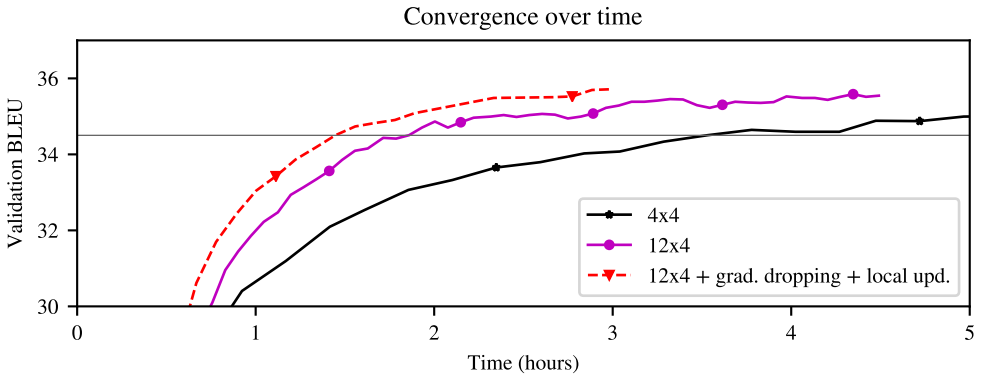
The approach of linearly scaling up the learning rate and scale down the warm-up period cannot be applied indefinitely. If the learning rate is too big, the model starts to overshoot and fails to converge. Similarly, if the warm-up period is too short, the model will be unstable as the learning rate increment is too steep. Prior work stated that finding the upper limit of the learning rate is not simple, and might be task/dataset/model dependent.

Orthogonally, our gradient dropping cannot be scaled indefinitely since summing more sparse gradients across nodes results in a less sparse global gradient, thus reducing the compression ratio. To retain a high compression ratio, we can re-sparsify the gradient back or simply increase the initial compression ratio. Both options compress more gradient, therefore potentially damaging the quality.

We scale the 4x4 multi-node experiment to 48 GPUs distributed across 12 nodes (12x4). Shard communication allows us to scale the node without increasing the communication cost; hence, 12x4 achieved a linear raw speed increase from 4x4 baseline,



(a) Convergence per-update



(b) Convergence over time

Figure 5.4: Convergence of Transformer model trained with 48 GPUs. Training is parallelized across 12 nodes, with 4 GPUs each.

as shown in Table 5.2.

However, even without additional tuning, the model trained with (12x4 nodes) has better convergence per update (Figure 5.4a), which we attribute to the larger batch size. Unfortunately, increasing the learning rate reduces the quality by -0.7 BLEU after multiplying the learning rate by 2 or more with a larger multiplier. However, increasing the warm-up period mitigates the quality damage—though the model converged slower. Therefore, we had to continue with the same learning rate as the 4x4 multi-node. Since we achieve a (nearly) linear raw speed increase, 12x4 reaches near-convergence BLEU 1.9x faster than the 4x4, as shown in Figure 5.4b and Table 5.2. While the improvement is certainly useful, it is not efficient since we spend 3x more computational power to gain less than 2x the benefit.

This experiment concludes that scaling the hyperparameters for multi-node training is challenging. Without scaling the hyperparameters (the learning rate in this case)

properly, the training speed improvement is sub-optimal. However, there is a limit to how high the learning rate can be set. This finding is congruent with Goyal et al. (2017).

We then apply a gradient dropping with local context to the 12x4 multi-node setting. Similarly, we drop 99% of gradients. With 12 nodes, the summed sparse gradient is less sparse. We empirically find that the average sparsity is reduced to 96%. Nonetheless, we still substantially improve the raw speed by 10.2x over the single-node and by 1.36x over the uncompressed 12x4 multi-node setting. There is no significant convergence degradation per update, as shown in Figure 5.4a.

In our experiment, the sparse gradient update is still usable in the 48 GPUs setting. Theoretically, with even more nodes, the summed sparse gradients will be dense enough, thus negating the speed increase. However, we argue that under larger-scale experiments scaling the hyperparameters properly should be prioritised above the sparse gradient updates, which should be considered for future studies. We see this as interesting future work.

5.6 Conclusion

We improve model convergence when training with sparse gradients by utilising an additional locally-computed gradient while also negating quality loss (in terms of BLEU) caused by gradient dropping. With gradient dropping and local gradient incorporation, we improve the raw training speed in terms of words/second by up to 3.7x (from the ideal case of 4x speed-up) over the single-node system, and by up to 1.4x over the uncompressed multi-node system. We also evaluate the training speed based on the time required to reach a near-convergence BLEU score. In this case, we improve the training speed by up to 3.5x (from the ideal case of 4x speed-up) over the single-node system and by up to 1.5x over the uncompressed multi-node system.

Chapter 6

Transfer Learning as a Better Initialization

In this chapter, we perform several black box ablation studies that limit information transfer and then measure the quality impact to gain an understanding of transfer learning. We observe how transfer learning acts to eliminate the warm-up phase in a transformer architecture. This chapter is based on Aji et al. (2020).

6.1 Introduction

Transfer learning, where a high-resource NMT model is transferred to initiate a low-resource model, is a common method for improving the low-resource NMT performance (Zoph et al., 2016; Dabre et al., 2017; Qi et al., 2018; Nguyen and Chiang, 2017; Gu et al., 2018b). However, it is unclear what settings make transfer learning successful and what knowledge is being transferred.

Understanding why transfer learning is successful can improve best practices while also opening the door to investigating ways to gain similar benefits without requiring parent models. In this paper, we perform several ablation studies on transfer learning in order to understand what information is being transferred.

We apply a black box methodology by measuring the quality of end-to-end translation systems. Typically, our experiments have a baseline that was trained from scratch, an off-the-shelf transfer learning baseline and simplified versions of the transfer learning scheme. If a simplified version recovers some of the quality gains of full transfer learning, it suggests that the simplified version has captured some of the information being transferred. Since information may be transferred redundantly, our claims are

limited to sufficiency rather than exclusivity.

Transferring word embeddings is not straightforward since languages have different vocabularies. Zoph et al. (2016) claimed that vocabulary alignment is not necessary, while Nguyen and Chiang (2017) and Kocmi and Bojar (2018) suggest a joint vocabulary. We find that the vocabulary has to be aligned before transferring the embedding to achieve a substantial improvement. Transfer learning without the embedding or with vocabulary mismatches is still possible, but with lower quality. Conversely, transferring only the word embeddings can be worse than transferring nothing at all.

A rudimentary model of machine translation consists of alignment and token mapping. We hypothesize that these capabilities are transferred across languages. To test this, we experiment with transferring from auto-encoders that learn purely diagonal alignment and possibly language modelling. To remove the effect of language modelling, we train auto-encoders on random strings sampled uniformly. However, all of these scenarios still have simple copying behaviour, especially with tied embeddings. Therefore, we also attempt a bijective vocabulary mapping from source to target, forcing the model to learn the mapping as well. Curiously, parents trained with bijectively-mapped vocabularies transfer slightly better to children.

We then investigate transfer learning for high-resource children, where the goal is reduced training time since they mainly attain the same quality. Transfer learning primarily replaces the warm-up period, though only real language parents yielded faster training.

6.2 Related Work

Transfer learning has been successfully used in low-resource scenarios for NMT. Zoph et al. (2016) gain 5 BLEU points in Uzbek–English by transferring from French–English. Their style of transfer learning copies the entire model, including word embeddings, ignoring the vocabulary mismatch between parent and child. They used separate embeddings for source and target language words, whereas tied embeddings (Junczys-Dowmunt et al., 2018) have since become the de-facto standard in low-resource NMT. Tied embeddings provide us with the opportunity to revisit some of their findings. In Section 6.5, we find an English–English copy model does work as a parent with tied embeddings, whereas Zoph et al. (2016) reported no gains from a copy model with untied embeddings.

Methods to cope with vocabulary mismatch have improved since Zoph et al. (2016).

Kocmi and Bojar (2018) suggest that a shared vocabulary between the parent language and the child is beneficial, though this requires knowledge of the child languages when the parent is trained. Addressing this issue, Gheini and May (2019) proposed a universal vocabulary for transfer learning. Their universal vocabulary was obtained by jointly training the sub-word tokens across multiple languages at once, applying Romanisation to languages in non-Latin scripts. However, unseen languages may only be representable in this universal vocabulary with a very aggressive and potentially sub-optimal subword segmentation. Orthogonally, Kim et al. (2018); Lample et al. (2018); Artetxe et al. (2018); Kim et al. (2019b) use bilingual word embedding alignment to initialise the embedding layer to tackle low resource language pairs. In Section 6.4.2, we compare a variety of vocabulary transfer methods.

Prior work (Dabre et al., 2017; Nguyen and Chiang, 2017) stated that a related language is the best parent for transfer learning. Lin et al. (2019) explore options to choose the best parent and conclude that the best parent language might not necessarily be related but is instead based on external factors such as the corpus size. In Section 6.3, we try two parent models in both directions to set baselines for the rest of the paper; an exhaustive search is not our main purpose.

Another approach to low-resource (or even zero-shot) NMT is through multilingual models (Johnson et al., 2016), which is similar to training the parent and child simultaneously. A related idea creates meta-models with vocabulary residing in a shared semantic space (Gu et al., 2018a,b).

If there is more parallel data with a third language, often English, then pivoting through a third language can outperform direct translation (Cheng et al., 2016). This approach requires enough source–pivot and target–pivot parallel data, which is arguably hard in many low resource scenarios, such as Burmese, Indonesian, and Turkish.

Orthogonal to transfer learning, Lample et al. (2018) and Artetxe et al. (2018) have proposed a fully zero-shot approach for low resource languages that relies on aligning separately-trained word embeddings to induce an initial bilingual dictionary. The dictionary is then used as the basis for a translation model. However, these methods do not generalise to arbitrary language pairs (Søgaard et al., 2018). Moreover, our setting presumes a small amount of parallel data in the low-resource pair.

6.3 Baseline Transfer Learning

We start with arguably the simplest form of transfer learning: train a parent model then switch to training with the child's dataset following Zoph et al. (2016). We attempt to match and initialise the embedding vectors of the same tokens from the parent to the child. We later investigate different approaches to transferring the embeddings. As transfer learning requires a parent model, we start by sweeping different high-resource languages for the parent model to set a baseline.

Choosing a parent language pair is one of the first issues to solve when performing a transfer-learning experiment. However, this is not a simple task. Prior work (Dabre et al., 2017; Nguyen and Chiang, 2017) suggest that a related language is the best option, albeit related is not necessarily well defined. Recently, Lin et al. (2019) performed a grid-search across various parent languages to determine the best criteria for selecting the optimal parent when performing transfer learning. Their work showed that the best language parents might also be determined by external factors such as the corpus size, on top of the language relatedness.

We first explore four potential parents: German and Russian from/to English. From each of them, we transfer the parameters to our low-resource language pair of {Burmese, Indonesian, Turkish} to English. Before presenting the results, we lay out the experimental setup used for the rest of the paper.

6.3.1 High-resource Datasets

We use German-English and Russian-English datasets for our parent models. Our German-English dataset is taken from the WMT17 news translation task (Bojar et al., 2017). Our Russian-English is taken from the WMT18 task (Bojar et al., 2018). For both pairs, we preprocess the input with byte-pair encoding (Sennrich et al., 2016c).

6.3.2 Low-resource Datasets

We use the following datasets:

Burmese–English: For our My→En parallel data, we used 18k parallel sentences from the Asian Language Treebank (ALT) Project (Ding et al., 2018, 2019) collected from news articles.

Indonesian–English: Id→En parallel data consists of 22k news-related sentences,

which are taken from the PAN Localization BPPT corpus.[1] This dataset does not have a test/validation split. Hence we randomly sample 2000 sentences to use as test and validation sets. We augment our data by backtranslating (Sennrich et al., 2016b) News Crawl from 2015. Our total training set (including the back-translated sentences) consists of 88k pairs of sentences.

Turkish–English: Tr→En data comes from the WMT17 news translation task (Bojar et al., 2017). This data consists of 207k pairs of sentences. Similar to Id→En, we add a back-translation corpus from News Crawl 2015. Our total training data consists of 415k sentence pairs.

For all language pairs, we use byte-pair encoding (Sennrich et al., 2016c) to tokenise words into subword units.

6.3.3 Training Setup

We use a standard transformer-base architecture with six encoder and six decoder layers for all experiments with the default hyper-parameters (Vaswani et al., 2017). Training and decoding use Marian (Junczys-Dowmunt et al., 2018), while evaluation uses SacreBLEU (Post, 2018).

6.3.4 Results

	BLEU		
Parent	My→En	Id→En	Tr→En
-	4.0	20.6	19.0
En→De	17.5	27.5	20.2
En→Ru	17.8	27.4	20.3
De→En	17.3	26.3	20.1
Ru→En	17.1	26.8	20.6

Table 6.1: Transfer learning performance across different language parents.

Our results on Table 6.1 show that there is no clear evidence that one parent is better than another. Whether the non-English languages share a script or English is on the same side does not have a consistent impact. The main goal of this section was to

[1] http://www.panl10n.net/english/OutputsIndonesia2.htm

set appropriate baselines; we primarily use English→German and German→English as the parents.

6.4 Transferring Embedding Information

Parent and child languages have a different vocabulary, so embeddings are not inherently transferable. We investigate what is transferred in the embeddings and evaluate several vocabulary combination methods.

6.4.1 Are the Embeddings Transferable?

We first explore whether the embedding matrix contains any transferable information. We divide the model into embedding parameters and everything else: inner layers. Table 6.2 shows what happens when these parts are or are not transferred.

Transferring		BLEU						
		De→En parent			En→De parent			
Emb.	Inner	My→En	Id→En	Tr→En	My→En	Id→En	Tr→En	avg.
Y	Y	17.8	27.4	20.3	17.5	27.5	20.2	21.7
N	Y	13.6	25.3	19.4	10.8	24.9	19.3	18.3
Y	N	3.0	18.2	19.1	3.4	18.8	18.9	13.7
N	N	4.0	20.6	19.0	4.0	20.6	19.0	14.5

Table 6.2: Transfer learning performance by only transferring parts of the network. Inner layers are the non-embedding layers. N = not-transferred. Y = transferred.

Our low-resource languages achieve better BLEU even if we only transfer the inner layers. In contrast, only transferring the embeddings is not beneficial, and sometimes it is even harmful to the performance. Finally, transferring all layers yields the best performance.

To further investigate which part of the network is more crucial to transfer, we took the best-performing child then reset either the embeddings or inner layers and restarted training. We explore whether the model is capable of recovering the same or comparable quality by retraining. We can look at this experiment as 'self' transfer learning. Results are shown in Table 6.3. When the inner layers are reset, self-transfer performs poorly (close to the quality without transfer learning at all), even though the

	BLEU		
Transfer	My→En	Id→En	Tr→En
baseline (no transfer)	4.0	20.6	19.0
transfer, train	17.8	27.4	20.3
transfer, train, reset emb, train	13.3	25.0	20.0
transfer, train, reset inner, train	3.6	18.0	19.1

Table 6.3: Investigating the model's capability to restore its quality if we reset the parameters. We use En→De as the parent.

embeddings are properly transferred. Conversely, the models can somewhat restore their quality even if we reset the embedding layer. This result further verifies that transferring the inner layers is the most critical aspect of transfer learning.

We conclude that transferring the inner layers is critical to performance, with far more impact than transferring the embeddings. However, the embedding matrix has transferable information, as long as the inner layers are included.

6.4.2 How to Transfer the Embeddings

Mixed recommendations exist on how to transfer embeddings between languages with different vocabularies. We compare methods from previous work, namely random assignment (Zoph et al., 2016) and joint vocabularies (Nguyen and Chiang, 2017) with two additional embedding assignment strategies based on the frequency and token matching as a comparison. In detail, we explore:

- **Exclude Embedding:** We do not transfer the embeddings at all. As such, we show that transfer learning works without transferring the embedding layer. In the present experiment, this method acts as one of the baselines.

- **Frequency Assignment:** We can transfer the embedding information regardless of the vocabulary mismatch. However, the toolkit sorts the words based on their frequency; therefore, embeddings are also transferred in that particular order. Regardless, we can determine whether word frequency information is transferred.

- **Random Assignment:** Zoph et al. (2016) suggest that randomly assigning a parent word embedding to each child word is sufficient, relying on the model to

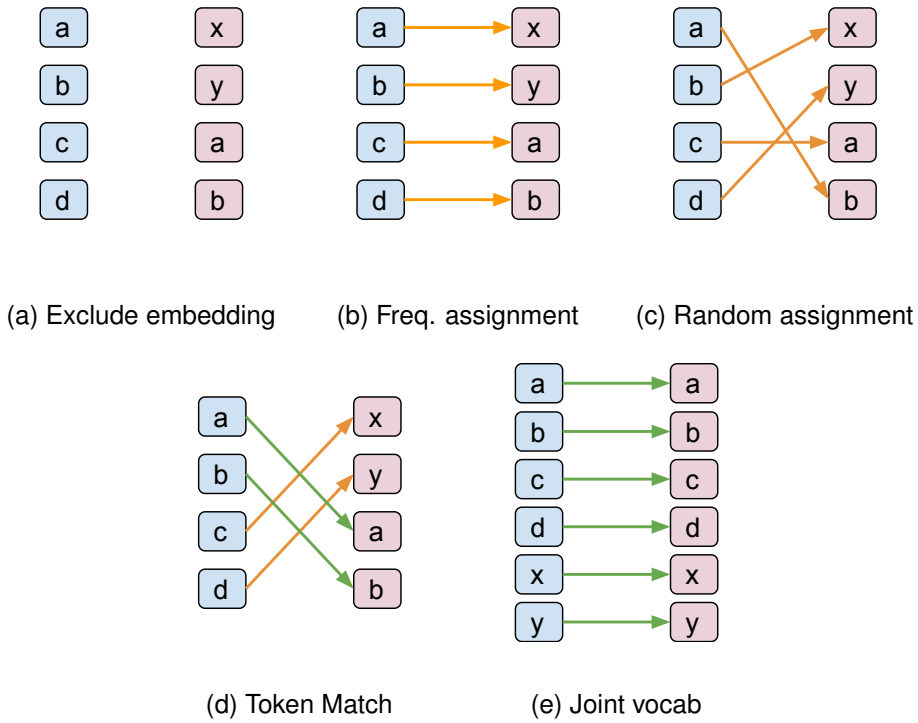


Figure 6.1: Illustration of various strategies on how to transfer the embedding vector.

untangle the permutation. This approach is simple and language-agnostic, thus universally applicable. We shuffle the vocabulary to achieve a random assignment.

- **Joint Vocabulary:** Nguyen and Chiang (2017) suggest that it is better to use a shared vocabulary between the parent and child language. This can be obtained by training a joint BPE token. To achieve this, we transfer the word embedding information of the common tokens. Since tied embeddings are used, we share the same vocabulary between the target and source of both the parent and the child language. One drawback of this technique is that we must prepare the vocabulary in advance. Therefore, switching the parent or the child might require us to re-train the model.

- **Token Matching:** We assign the embeddings with the same token first and randomise the rest. This approach is designed to allow some word embeddings to be transferred correctly without the need to re-train the parent with every experiment, as in the case of joint vocabulary.

The different strategies are illustrated in Figure 6.1.

	BLEU						
	De→En parent			En→De parent			
Embedding	My→En	Id→En	Tr→En	My→En	Id→En	Tr→En	avg.
-	4.0	20.6	19	4.0	20.6	19	14.5
Exclude embedding	13.6	25.3	19.4	10.8	24.9	19.3	18.3
Frequency assign	14.2	24.4	19.4	13.9	24.3	19.4	19.2
Random assign	13.9	24.6	19.2	13.8	23.9	19.3	19.0
Token matching	17.8	27.4	20.3	17.5	27.5	20.2	21.7
Joint vocabulary	18.5	27.5	20.9	18.5	28.0	19.6	22.0

Table 6.4: Transfer learning performance with different ways to transfer the embedding.

Prior experiments in Section 6.4.1 demonstrate that we can apply transfer learning even if we only transfer the inner layers. Curiously, random assignment and frequency assignment are not better than excluding the embeddings, except for Burmese to English transferred from English to German. Therefore, the information in the embedding is lost when transferred to the incorrect token. From these results, we conclude that the model is incapable of untangling the embedding permutation as stated by Zoph et al. (2016).

Transfer learning yields better results when we attempt to transfer the embeddings to the correct tokens. In the joint vocabulary setting, not every token is observed in the parent language dataset; therefore, only a section of the embedding layer is correctly trained. However, we still observe a significant improvement over the random and frequency-based assignment.

We can also transfer the embedding vectors by matching and assigning the word embedding with the same tokens. Vocab matching achieves comparable results to joint vocabulary, except for the lowest-resource language, Burmese. Therefore, this simple matching can be used as a cheaper alternative over a joint vocabulary. On top of that, this approach is more efficient as we do not transfer and wastefully reserve extra memory for tokens that will not be seen in the child language.

These results suggest that word information stored in the embedding layer is transferable, as long as the vectors are assigned correctly. Therefore, better ways of handling the embedding layer transfer are joint BPE and token matching, as they further improve the performance of the child language pair.

6.5 Transferring Structural Information

Parent	Shared	Example
En→De	Id→En	src: Bank Mandiri bisa masuk dari mikro hingga korporasi . out: Bank Mandiri bisa memperingatkan dari cen@@ hingga korporasi . alignment: 0-0 1-1 3-3 5-5 6-6 7-7 9-2 9-4 9-8 9-9
De→En	Id→En	src: Bank Mandiri bisa masuk dari mikro hingga korporasi . out: seperti Mandiri bisa masuk a mikro hingga korporasi . alignment: 2-2 3-0 3-1 3-3 3-9 5-5 6-6 7-7 7-8 9-4

Table 6.5: Output example of transferred model without fine tuning. The model performs monotonic alignment.

To understand what information is being transferred with transfer learning, we test the parent model's performance on the child language without any additional training.

When a pre-trained model is transferred to another language pair, the model has not yet seen the child language vocabulary. When presented with an input in a new language, the model is unable to translate correctly. However, as we can see in Table 6.5, the model manages to perform diagonal alignment properly, albeit it is mostly copying the input (on average of 75% of the time).

Based on this observation, we see that fallback copying behaviour, including monotonic alignment, is transferred. This can be useful for named entity translation (Currey et al., 2017). To test our claim, we prepare parents that implicitly learn to copy or transform input tokens diagonally.

We can create a copy sequence model (or auto-encoder) model by giving the model the same sentences for both source and target. We pick an English monolingual dataset. We also use a Chinese monolingual corpus to explore whether the chosen monolingual language matters. Besides, we can artificially create a random sequence for the training set. The random sequence is useful to determine whether any language-specific information is being transferred, as such information is absent in a random sequence.

To simulate the translation behaviour better, we also prepare a substitution parallel corpus. We transform every token into another based on a predetermined 1:1 mapping. We create a substitution corpus for both the English and the synthetic corpus. With tied embeddings, the substitution corpus should help the model translate one token into another, instead of just copying. Table 6.6 illustrates the 6 monolingual/synthetic parents that we use for this experiment.

Parent	Type
Mono copy sequence	**src:** Madam President , on a point of order .
(En→En)	**tgt:** Madam President , on a point of order .
Mono sub sequence	**src:** Click write , ideologies rotate sful ECHO recommended struggle
(En_S →En)	**tgt:** Madam President , on a point of order .
Mono copy sequence	**src:** 保持点神秘感。
(Zh→Zh)	**tgt:** 保持点神秘感。
Mono sub sequence	**src:**比赛漂亮家宝1503 知识产权
(Zh_S →Zh)	**tgt:** 保持点神秘感。
Random copy sequence	**src:** 1 3 2 1 1
(Rand→Rand)	**tgt:** 1 3 2 1 1
Random sub sequence	**src:** 2 4 3 2 2
($Rand_S$ →Rand)	**tgt:** 1 3 2 1 1

Table 6.6: Monolingual and random parents with their sentence example.

We perform transfer learning experiments from every monolingual and synthetic parent to all three child languages, as summarised in Table 6.7. For comparison, we also provide the result of transfer learning with an actual translation model as a parent. We notice that there is no improvement in transfer learning for the Turkish model in terms of the final BLEU. However, upon further investigation, transfer learning has an impact on the convergence speed, thus signalling information being transferred. To measure this, we capture the validation BLEU score for Tr→En after 10k training steps.

In general, transferring from any monolingual or synthetic parent yields better BLEU (or faster convergence for Turkish) compared to training from scratch. Although, the improvement is sub-optimal when compared with transfer learning from a proper parent. However, we can use these gains to measure the information transferred in transfer learning.

In general using monolingual English is better than using monolingual Chinese. In monolingual English, we can transfer the embedding information correctly with token matching. Therefore, consistent with our previous experiment, embedding information is transferred.

Using a Chinese parent is better than using random sequences. Our random sequence is uniformly sampled independently for each token. Therefore, unlike a real monolingual corpus, learning language modelling from this random sequence is im-

	BLEU			
Parent	My→En	Id→En	Tr→En	Tr(10k)
-	4.0	20.6	19.0	14.3
De→En	17.8	27.4	20.3	20.2
En→En	10.4	23.3	18.5	16.0
En_S →En	12.3	23.8	19.0	16.5
Zh→Zh	8.3	22.5	18.8	16.3
Zh_S →Zh	11.2	23.5	19.0	16.3
Rnd→Rnd	6.2	21.9	19.0	15.2
Rnd_S →Rnd	7.9	22.0	19.3	15.1

Table 6.7: Transfer learning performance on monolingual and synthetic parents. We also measure the validation BLEU of Tr→En after 10k updates.

possible. Thus, we conclude that the model transfers some statistical properties of natural languages.

Transferring from a random sequence copy model yields better result compared to training the model from scratch. While the improvement is minimal, we can see that a naïve model that performs copying is better as a model initialisation. Substitution sequence parent models perform better than their copying counterparts only on Burmese. We conclude that alignment is transferred.

Transfer learning with an actual NMT system as a parent still outperforms the monolingual and synthetic parents, albeit they are initially a copy model. We argue that the monolingual parents perform nearly perfectly at the copying task, and have perfect diagonal alignment, and therefore overfit to this artificial setting when used as a parent.

6.6 Transfer Learning for High-Resource Languages

Transfer learning can be used to initialise a model even if final quality does not change. Compared to random initialisation, we argue that a pre-trained model functions as better initialisation. Therefore, since we initialise the model better, it should converge faster. This behaviour was already presented in Table 6.7, where the transferred model converges more rapidly. However, we should explore this behaviour in a setting where faster training matters more: when training high-resource language pairs.

For this experiment, we take an English-to-Russian model as a parent for an English-to-German model. We align the embedding with the same BPE tokens instead of using a joint vocabulary since this would require re-training the parent. We also attempt to exclude the embedding completely. These choices are practical in a real-world scenario, especially when we measure for efficiency.

Parent	BLEU	Num. Steps to 34 BLEU
Baseline	35.6	48k
+ no warm-up	0.0	-
En→En$_S$	35.4	60k (0.8x faster)
En→Ru	35.7	40k (1.2x faster)
+ token matching	35.7	34k (1.4x faster)
+ no warm-up	35.6	22k (2.1x faster)

Table 6.8: Transfer learning effect to the model's quality of high-resource language. We also measure the time to reach a near-convergence level of 34 BLEU.

In Table 6.8, we show that transfer learning does not improve the model's final quality. However, we can see both from the Table, and visually in Figure 6.2, that transfer learning speeds up the convergence by up to 1.4x, assuming the parent model has been prepared before.

In the early stage of training, the gradients produced are quite noisy, which is particularly harmful to the transformer model (Popel and Bojar, 2018). Therefore, training transformer models usually require a precise warm-up setup. However, transfer learning can be used as a better initialisation, thus skipping the noisy early training. To further confirm this, we remove the learning rate warm-up to observe the impact of a pre-trained model.

As shown in Figure 6.2, the pre-trained model remains capable of learning under more aggressive hyperparameters. On the other hand, the model without pre-training fails to learn. This result is congruent with the findings of Platanios et al. (2019), who found that warm-up in the Transformer can be removed with curriculum learning.

6.7 Conclusion

We demonstrate that the internal layers of the network are the most crucial for cross-lingual transfer learning. The embeddings contain transferable information, as long as

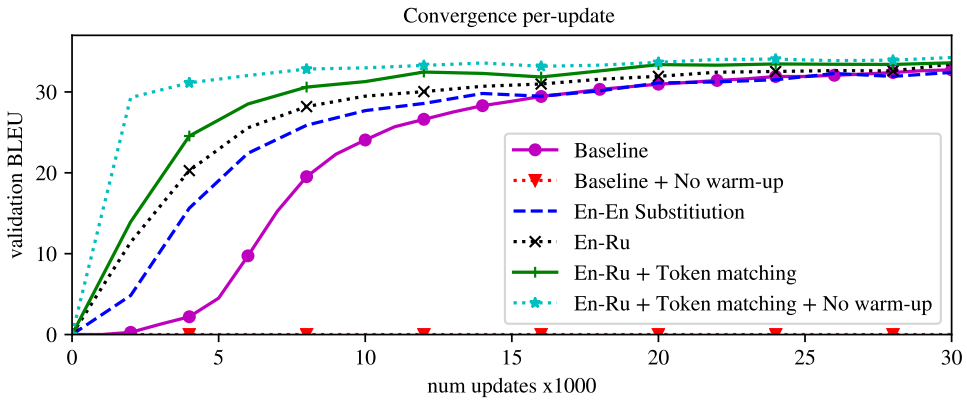


Figure 6.2: Transfer learning effect on the convergence of a high-resource system. Transfer learning removes the need for warm-up.

the vectors are mapped correctly and the inner layers are also transferred. While not as optimal, we can still perform transfer learning by excluding the embedding. In transfer learning, we can also transfer the alignment. Transferred parents without fine-tuning will align the input diagonally and copy most of the tokens. We further demonstrate that transfer learning still functions with a simple copy model, even with an artificial dataset—albeit with a reduced quality.

From a theoretical perspective, our results indicate that while transfer learning is effective in our scenario, it performed less "transfer" than previously thought. Therefore, a promising research direction to investigate would involve the development and assessment of improved initialisation methods that would more efficiently yield the benefits of the model transfer.

From a practical perspective, our results indicate that we can initialise models with a pre-trained model regardless of the parent language or vocabulary handling. With this perspective in mind, we can use transfer learning as a better initialisation, resulting in the child model having more stable gradients from the onset of training. Therefore, models can train and converge faster, which is useful in high-resource settings. With transfer learning, the model can be trained with more aggressive hyperparameters—such as removing the learning rate warm-up entirely—to further improve the convergence speed. This result further highlights the use of transfer learning as a better model initialisation.

Chapter 7

4-bit Transformer Model

Previous chapters have focused on more efficient training of NMT via reducing communication and better model initialisation. In this chapter, we will focus on the more efficient deployment of NMT models via model compression. We design a quantisation procedure to compress NMT models better for devices with limited hardware capability. Since most neural network parameters are near zero, we employ logarithmic quantisation instead of fixed-point quantisation. This chapter is based on Aji and Heafield (2019b).

7.1 Introduction

Neural Machine Translation (NMT) is resource-demanding. Current state-of-the-art architectures, such as the Transformer (Vaswani et al., 2017) or deep RNN (Miceli-Barone et al., 2017) are typically hundreds of megabytes in size. In a client-based translation system, these large models must be deployed locally, thus consuming network bandwidth for distributing the model, and disk space for storing the model.

Model quantisation has been widely studied as a way to compress model size and increase the inference speed. However, most of this work has focused on convolution neural networks for computer vision tasks (Miyashita et al., 2016; Lin et al., 2016; Hubara et al., 2016, 2017; Jacob et al., 2018). As such, research on model quantisation for NMT tasks remains limited.

We find that the model can be compressed at up to 4-bit precision without sacrificing quality. We first explore the use of logarithmic-based quantisation over fixed-point quantisation (Miyashita et al., 2016) based on the empirical findings that parameter distribution is not uniform, but instead concentrated near zero (Lin et al., 2016; See

et al., 2016). The magnitude of a parameter also varies across layers; therefore, we propose an improved method of scaling the quantization centres. We also notice that biases do not quantise very well. However, since biases do not consume a noticeable amount of memory, they can be left unquantised. Lastly, we explore the significance of re-training in the model compression scenario. We adopt an error feedback mechanism (Seide et al., 2014) to preserve the quantisation error rather than discarding it at every update during re-training.

7.2 Related Work

A considerable amount of research on model quantisation has been performed in the area of computer vision with convolutional neural networks; however, research on model quantisation in the field of neural machine translation is far more limited. Therefore, we will also refer to work on neural models for image processing in this section, where appropriate.

Hubara et al. (2016) quantised the model and activation to binary on a CNN network for various image classification tasks. The binary network achieved near state-of-the-art quality on several easier tasks such as MNIST and CIFAR-10 but achieved sub-par performance on the more challenging ImageNet dataset (losing over 20% accuracy with quantised GoogleNet). Hubara et al. (2017) later reported that with 6-bit fixed-point quantisation, GoogleNet "only" lost 5% of accuracy. Lin et al. (2016) used different bit precisions on various CNN layers, achieving over 20% compression on the CIFAR-10 task.

Since the model's parameters are highly concentrated near zero, Miyashita et al. (2016) opted for logarithmic quantisation. They report an improvement in preserving model accuracy over linear quantisation while achieving the same model compression rate. They also reported negligible accuracy degradation when compressing VGG16 with 3-bit logarithmic quantisation, whereas 3-bit fixed-point quantisation suffered a 6% accuracy drop.

Hubara et al. (2017) compress an LSTM-based architecture for language modelling to 4-bits without any quality degradation but had to scale the hidden layer size by 3. See et al. (2016) pruned an NMT model by removing any weight values lower than a certain threshold. They achieve 80% model sparsity without any quality degradation.

A relevant work with respect to our purposes is the submission of Junczys-Dowmunt et al. (2018) to the Shared Task on Efficient Neural Machine Translation in 2018. This

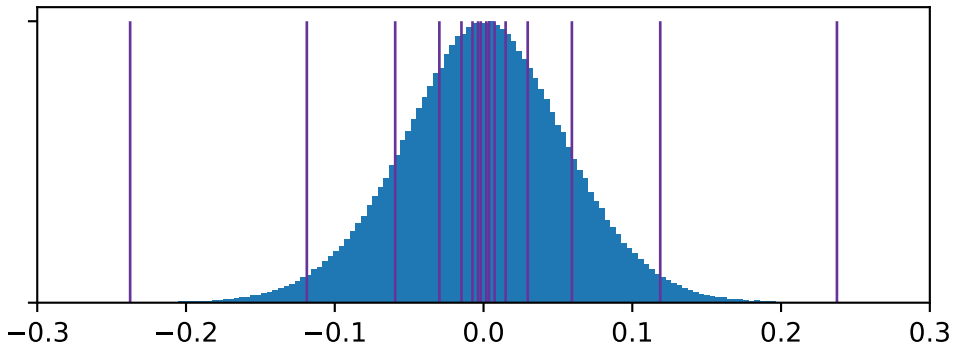


Figure 7.1: Histogram of the first encoder's feed-forward network weight. Parameters follow a normal distribution. Vertical lines illustrate the log-based quantisation centres.

submission applied an 8-bit linear quantisation for NMT models without any noticeable deterioration in translation quality. Similarly, Quinn and Ballesteros (2018) proposed the use of 8-bit matrix multiplication to increase the CPU inference speed of an NMT system.

7.3 Low-precision Neural Machine Translation

7.3.1 Log-based Compression

Parameters in deep learning models are normally distributed Lin et al. (2016); See et al. (2016). Therefore, a uniformly distributed fixed-point quantisation may not fit the parameter distribution. To improve resolution for small values, we adopt logarithmic quantisation following Miyashita et al. (2016) where parameter density is the highest. Figure 7.1 illustrates the weight distribution and our log-based quantisation.

We use the same quantisation centres for positive and negative values. When compressing to B bits, a single bit represents the sign while the remaining $B-1$ bits represent the log magnitude. The centres are tuned based on the absolute value of the data.

For efficient implementation and because the impact on quality was minimal after re-training, we use log base 2. Log base 2 means that exponentiation amounts to a bit-shift while taking a rounded log (which will be used to quantise a value) is equivalent to finding the leftmost 1 in binary.

We find that tensors might not have the same parameter magnitude. Therefore we also scale the quantisation centres to approximate each tensor better. This approach is

different from that of Miyashita et al. (2016), where quantisation centres are not scaled, thus letting every tensor to have the same set of centres. Formally, each quantisation centre takes the form $\pm S2^q$ where S is a scaling factor, and q is an integer in the range $(-2^{B-1}, 0]$. The scaling factor S is selected separately for each tensor in the model.

To minimise the mean squared encoding error, values should be quantised to the nearest centre. Miyashita et al. (2016) find the nearest centre in logarithmic space by taking the log and then rounding to the nearest integer, which is not the same as finding the nearest centre in normal space. For example, their approach will quantise 5.8 to 2^3 instead of 2^2 because $\log_2(5.8) \approx 2.536$, which rounds to 3. In normal space, 5.8 is closer to 2^2 instead of 2^3.

We can implement rounding to the nearest centre in normal space efficiently by multiplying by $\frac{2}{3}$, taking the log and rounding up to the next integer. Let $x \in [2^q, 2^{q+1}]$. Thus:

$$\begin{aligned} x \text{ rounds up to } 2^{q+1} &\iff x > \frac{2^q + 2^{q+1}}{2} \\ &\iff x > \frac{2^q(1+2)}{2} \\ &\iff \frac{2}{3}x > 2^q \\ &\iff \log_2 \frac{2}{3}x > q \end{aligned} \tag{7.1}$$

Therefore, given a positive x, we can find the quantised magnitude of q with respect to rounding scheme in normal space by:

$$q = \lceil \log_2(\frac{2}{3}t) \rceil \tag{7.2}$$

Ultimately, given a value v that will be quantised a B-bit logarithmic quantisation. We encode v as $(sign, q)$, where $sign$ represents the sign (1-bit), and q represents the magnitude ($B-1$ bits). Our quantisation functions as follows:

$$\begin{aligned} sign &= sign(v) \\ t &= clip(|v|/S, [1, 2^{1-2^{B-1}}]) \\ q &= \lceil \log_2(\frac{2}{3}t) \rceil \end{aligned} \tag{7.3}$$

where t is a temporary variable. We first scale the value to the desired range based on scaling factor S. We will discuss more on computing S later. Then, we clip the value

into the given range since we have limited quantisation centres. This then decodes to $v' \approx v$ as $v' = \text{sign}S2^q$. In practice, the sign is stored with q.

7.3.2 Selecting the Scaling Factor

There are a few heuristics to choose a scaling factor of S. Junczys-Dowmunt et al. (2018) and Jacob et al. (2018) scale the model based on its maximum value, which can be very unstable–especially during re-training. Alternatively, Lin et al. (2016) and Hubara et al. (2016) use a pre-defined step size for fixed-point quantization. Our objective is to select a scaling factor S such that the quantised parameters are as close to the original as possible. Therefore, we optimise S such that it minimises the squared error between the original and the compressed parameters.

We start with an initial scale S based on the parameters' maximum value. For a given S, we apply our quantisation routine described in Equation 7.3 to a tensor v, resulting in an approximation of v'. For a given assignment v', we fit a new scale S such that:

$$S = \arg\min_S \sum_i (v'_i - v_i)^2 \tag{7.4}$$

Substituting v'_i within Eq. 7.4, we have:

$$S = \arg\min_S \sum_i (sign(v_i)S2^{q_i} - v_i)^2 \tag{7.5}$$

To simplify the equation, let a temporary variable a_i to substitute $sign(v_i)2^{q_i}$. Hence we have:

$$S = \arg\min_S \sum_i (a_i S - v_i)^2 \tag{7.6}$$

To optimise the given objective, we take the first derivative of Equation 7.6 such that:

$$
\begin{aligned}
\frac{d}{dS}\sum_i (a_i S - v_i)^2 &= 0 \\
2\sum_i (a_i(a_i S - v_i)) &= 0 \\
\sum_i (a_i^2 S) - \sum_i (a_i v_i) &= 0 \\
S\sum_i a_i^2 &= \sum_i (a_i v_i) \\
S &= \frac{\sum_i (a_i v_i)}{\sum_i {a_i}^2} \\
S &= \frac{\sum_i (sign(v_i) 2^{q_i} v_i)}{\sum_i (sign(v_i) 2^{q_i})^2} \\
S &= \frac{\sum_i (2^{q_i} |v_i|)}{\sum_i 4^{q_i}}
\end{aligned}
\tag{7.7}
$$

We optimise S for each tensor independently.

7.3.3 Re-training

We observe later in Section 7.4.2 that quantisation damages the model. Therefore, we re-train the model after initial quantisation to allow it to recover some of the quality loss. In the re-training phase, we compute the gradients normally with full precision. We then re-quantise the model after every update to the parameters, including fitting scaling factors.

Re-quantising the model after every update introduces quantisation errors. The re-quantisation error is preserved in a residual variable and added to the next step's parameter (Seide et al., 2014) before quantisation. Essentially, this is the same error feedback mechanism that we introduced in Chapter 4 to reduce the impact of compression errors by preserving compression errors as stale gradient updates for the next batch. We find that re-training fails to work without this mechanism (Section 7.4.2).

7.3.4 Handling Biases

We do not quantise bias values in the model. We find that biases are not as highly concentrated near zero when compared to other parameters. Empirically, in our pre-trained Transformer architecture, bias has a higher standard deviation of 0.17 (compared to

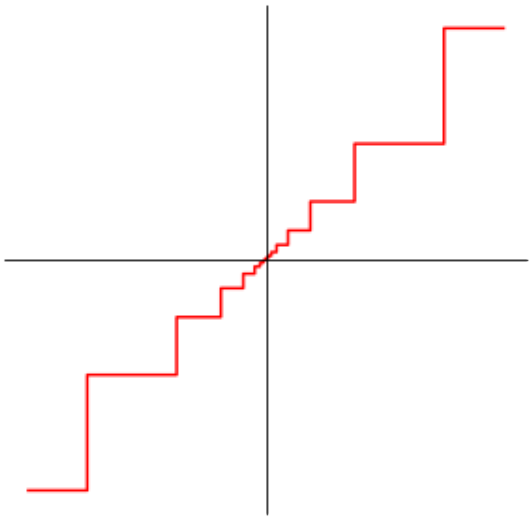


Figure 7.2: Log-quantization step function.

0.07 for other parameters). Attempting to log-quantise them used only a fraction of the available quantisation points. In any case, bias values do not consume a lot of memory relative to other parameters. In our Transformer architecture, they account for only $\sim$0.2% of the parameter values.

7.3.5 Low-precision Dot Products

Matrix multiplication operation is expensive. To improve the CPU inference speed, we explore training and computing dot products inside matrix multiplications in low precision. Activations coming into a matrix multiplication are quantised on the fly, while intermediate activations (such as tanh) are not quantised.

We use the same log-based quantisation procedure described in Section 7.3.1 when training the model. However, we only attempt a fixed pre-determined scale. Running the slower EM approach to optimise the scale before every dot product would not be fast enough for inference applications.

Training with Quantised Dot Products

Our log-quantised activation is a step function, as illustrated in in Figure 7.2. Therefore, the derivative of this function is 0 almost everywhere, or undefined in the quantization centres. Thus, we cannot back-propagate through this function normally. Inspired by Hubara et al. (2017), we utilise a straight-through estimator (Bengio et al., 2013) to set the derivative of the the function to 1, thus enabling training.

Computing Dot Products in Log-space

A dot product operation consists of two sub-operations: element-wise multiplication and sum. In our case, we now have two vectors a and b, both in the form of:

$$a = S_a * [(sign_{j1} * 2^{j_1}), \ldots, (sign_{jn} * 2^{j_n})]$$
$$b = S_b * [(sign_{k1} * 2^{k_1}), \ldots, (sign_{kn} * 2^{k_n})]$$

Multiplication is performed by adding the powers. We then add the resulting multiplications together normally, as follows:

$$a \cdot b = S_a * S_b \sum_i (sign_{ji} * sign_{ki} * 2^{j_i + k_i}) \tag{7.8}$$

Computing power is obtained by using a bit-shift, while computing $sign_{ji} * sign_{ki}$ can be performed using bitwise xor, therefore avoiding expensive multiplication instructions (Miyashita et al., 2016).

7.4 Experiments

7.4.1 Experiment Setup

We use systems for the WMT 2017 English-to-German news translation task for our experiment, which differs from the WNGT shared task setting previously reported. We use back-translated monolingual corpora (Sennrich et al., 2016b) and byte pair encoding (Sennrich et al., 2016c) to pre-process the corpus. Quality is measured based on BLEU (Papineni et al., 2002a) score using sacreBLEU script (Post, 2018).

We first pre-train baseline models with both Transformer and RNN architectures. Our Transformer model consists of six encoder and six decoder layers with tied embedding. Our deep RNN model consists of eight layers of bidirectional LSTM. Models were trained synchronously with a dynamic batch size of 40 GB per batch using the Marian toolkit (Junczys-Dowmunt et al., 2018). The models are trained until we observe no improvement in 10 consecutive validations. Models are optimised with the Adam optimiser (Kingma and Ba, 2014). The rest of the hyperparameter settings on both models follow the suggested configurations (Vaswani et al., 2017; Sennrich et al., 2017). We use wmt2016 as the test set.

7.4.2 4-bit Transformer Model

In this experiment subsection, we explore different ways to scale the quantisation centres, the significance of quantising biases and the significance of re-training. We use a pre-trained Transformer model as our baseline and apply our quantisation algorithm on top of that. This experiment focuses solely on the compression ratio. Therefore, models are decompressed back into a 32-bit floating-point value for inference.

Method	**Scaling**		
	Unscaled	**Max**	**Optimized**
32-bit model (Baseline)	35.66	-	-
Without retraining			
4-bit model	25.20	28.08	33.33
4-bit model + 32-bit bias	34.16	34.29	34.31
With retraining			
4-bit model	34.92	34.81	35.26
4-bit model + 32-bit bias	35.09	35.25	35.47

Table 7.1: 4-bit Transformer quantisation performance for English-to-German translation, measured in BLEU score. We explore different methods of determining the scaling factor as well as skipping bias quantisation and re-training.

Table 7.1 summarises the results. Using a simple (albeit unstable) max-based scaling has shown to perform better than not using the scale factor. However, fitting the scaling factor to minimise the quantisation squared error produces the best quality. The BLEU score differences between methods of choosing the scaling factor are diminished after re-training.

We can also see improvements by not quantising biases, especially without re-training. Without any re-training involved, we reached the highest BLEU score of 35.47 by using an optimised scale in addition to uncompressed biases. Without bias quantisation, we obtained a $\sim$7.9x compression ratio (instead of 8x) with a 4-bit quantisation. Based on this trade-off, we argue that it is more beneficial to keep the biases in full precision.

Re-training has shown to generally improve quality. After re-training, the quality differences between various scaling and biases quantisation configurations are minimal. These results suggest that re-training helps the model to fine-tune under a new quantised parameter space.

Training Routine

We prepare our 4-bit quantisation model by re-training from a full precision model. We also store the quantisation errors to be considered for the next update. In this subsection, we answer the question of whether it is necessary to perform these steps. We explore the preparation of the 4-bit model if trained from scratch. Similarly, we explore 4-bit model preparation without an error feedback mechanism. For this experiment, we use optimised scaling and 32-bit bias when applying 4-bit log quantisation.

Method	Fine-tune?	Error-feedback	Transformer	RNN
Baseline	-	-	35.66	34.28
4-bit log	✓	✓	35.47 (-0.19)	34.22 (-0.06)
4-bit log	✓	✗	34.45 (-1.21)	33.32 (-0.96)
4-bit log	✗	✓	28.54 (-7.12)	28.45 (-5.83)
4-bit log	✗	✗	0.05(-35.61)	0.00(-34.48)

Table 7.2: The model performance (based on BLEU score) of various training scenarios using both Transformer and RNN architectures

The results in Table 7.2 indicate that fine-tuning from a pre-trained model and error feedback are necessary to produce a high-quality 4-bit model. Removing either of them degrades the quality. BLEU score is dramatically reduced if we train the model from scratch. Likewise, the quantised model is practically unable to learn without the error feedback mechanism. As shown in Table 7.1, the quantised model achieved a 34.31 BLEU score without re-training. Re-training said model barely improves the BLEU to 34.45 without the error feedback mechanism.

Size Comparison

To demonstrate the improvement of our method, we compare several compression approaches to our 4-bit logarithmic quantisation method with re-training and without bias quantisation. One of the arguably naive methods used to reduce model size is the use of smaller unit size. For Transformer, we set the feed-forward dimension to 512 (from 2048) and the embedding size to 128 (from 512). For RNN, we set the dimension to 320 (from 1024) and the embedding size to 160 (from 512). Using this method, the model size is $\sim$8x smaller and similar to 4-bit quantisation in terms of the model compression rate.

Method	Transformer	RNN
Baseline	35.66	34.28
Reduced Dimension	29.03 (-6.63)	30.88 (-3.40)
4-bit fixed point	34.61 (-1.05)	34.05 (-0.23)
4-bit log (Ours)	35.47 (-0.19)	34.22 (-0.06)

Table 7.3: The model performance (based on BLEU score) of various quantisation approaches using both Transformer and RNN architecture.

We also introduce the 4-bit fixed-point quantisation approach as a comparison, which is based on Junczys-Dowmunt et al. (2018). However, we made a few modifications to the original approach. Firstly, we apply re-training, which is absent in their implementation. Moreover, we skip bias quantisation. Finally, we optimise the scaling factor instead of the suggested max-based scale.

Table 7.3 summarises the results, which indicate that reducing the model size by simply reducing the dimension resulted in the worst performance. Our result is in line with (Huang et al., 2019), who show that reducing the model size by using fewer layers degrades quality. Logarithmic-based quantisation has been shown to perform better when compared to fixed-point quantisation using both architectures.

The RNN model seems to be more robust towards the compression. RNN models exhibit reduced quality degradation in all compression scenarios. We hypothesise that the gradients computed with a highly compressed model are very noisy, thus resulting in noisy parameter updates. Our finding is in line with prior research, as well as in previous chapters that state Transformer is more sensitive towards noisy training conditions (Chen et al., 2018).

7.4.3 Quantised Dot-Product

Quality Benchmark

We now apply logarithmic quantisation for all matrix multiplication inputs. We use the same quantisation procedure as the parameter. However, we do not fit the scaling factor since it is very inefficient. Hence, we do not scale the quantization centres for the activation. For the parameter quantisation, we use an optimised scale with uncompressed biases based on the previous experiment. Table 7.4 presents the quality results of the experiment. Generally, we observe quality degradation compared to a

full-precision dot product.

Method	Transformer	RNN
Baseline	35.66	34.28
+ Model Quantisation	35.47 (-0.19)	34.22 (-0.06)
+ Dot Product Quantisation	35.05 (-0.61)	33.12 (-1.16)

Table 7.4: Model performance (in BLEU) of model quantisation with dot product quantisation using both Transformer and RNN architecture.

Speed Benchmark

Dot-Product Method	time (ns)
32-bit float	8.45699
8-bit integer	2.08390
4-bit log quantisation (16-bit Shift)	3.89595
4-bit log quantisation (8-bit Lookup table)	2.51924

Table 7.5: Time measurement of dot products of 128 elements with different value representations. We use a Cascade Lake processor.

Unfortunately, current hardware does not support a 4-bit instruction, thus our dot-product must be emulated using instructions with wider bit widths.[1]

Since there is no 4-bit or 8-bit shift instruction, we emulate 2^q in 16-bit instead. Alternatively, we can choose a lower base, for example $256^{\frac{1}{14}}$ instead of 2 so that the resulting power fits in 8-bit precision. In this case, we can use the 8-bit lookup table instruction `vpshufb` instead.

We benchmark our result with an 8-bit integer dot product based on the `vpdpbusds` instruction (which was introduced in the Cascade Lake to optimise 8-bit matrix multiplication) and a basic 32-bit float dot product using fused multiplication and addition.

Table 7.5 reports the time required to perform a dot product under different quantisation schemes. 8-bit lookup table is faster than 16-bit. Unfortunately, our 4-bit dot product is inefficient, resulting in it being much slower than an 8-bit dot product. With current hardware, the main advantage over 8-bit quantization is smaller model size, which is of interest for local deployment on mobile devices. Should future hardware

[1] https://github.com/kpu/intgemm/blob/log4-unstable/log4/log4.h

also support 4-bit instructions natively, 4-bit models could also improve decoding efficiency.

7.4.4 Beyond 4-bit precision

With 4-bit quantisation and uncompressed biases, we obtain a 7.9x compression rate. Bit width can be set below 4 bit to achieve an even better compression rate, albeit introducing more compression error. To explore this, we sweep several bit widths. Moreover, we skip bias quantisation and optimise the scaling factor.

Bit	**Transformer**		**RNN**	
	Size (rate)	**BLEU(Δ)**	**Size (rate)**	**BLEU(Δ)**
32	251 MB	35.66	361 MB	34.28
4	32 MB (7.88x)	35.47 (-0.19)	46 MB (7.90x)	34.22 (-0.06)
3	24 MB (10.45x)	34.95 (-0.71)	34 MB (10.49x)	34.11 (-0.17)
2	16 MB (15.50x)	33.40 (-2.26)	23 MB (15.59x)	32.78 (-1.50)
1	8 MB (30.00x)	29.43 (-6.23)	12 MB (30.35x)	31.71 (-2.51)

Table 7.6: Compression rate and performance of both Transformer and RNN with various bit widths. The compression rate between Transformer and RNN is not equal since they have different biases to parameter size ratio.

Training an NMT system below 4-bit precision remains a challenge. As shown in Table 7.6, model performance degrades with fewer bits being used. While this result might be acceptable, we argue that the result can be improved. One worthwhile idea would be to increase the unit size in an extremely low-precision setting. We have shown that 4-bit precision performs better compared to the full-precision model with (near) 8x compression rate. Moreover, Han et al. (2015) demonstrated that 2-bit precision image classification can be achieved by scaling the parameter size. An alternative approach is to have different bit widths for each layer (Hwang and Sung, 2014; Anwar et al., 2015).

We also observe the robustness of RNN over Transformer in this experiment since RNN models degrade less compared to the Transformer counterpart. The RNN model outperforms Transformer when compressing at binary precision.

7.5 Conclusion

We compress the model size in neural machine translation to approximately 7.9x smaller than 32-bit floats by using a 4-bit logarithmic quantisation. Bias terms can be left uncompressed without significantly affecting the compression rate. We also find that re-training after quantisation is necessary to restore the model's performance.

Matrix multiplication can further be quantised, although quality is sacrificed. Unfortunately, 4-bit dot products found in matrix multiplication are slow because current hardware does not natively support the necessary 4-bit instructions.

Chapter 8

Conclusion and Future Work

8.1 Conclusion

This thesis has focused on efficiency for NMT training and deployment, using a range of techniques such as asynchronous training, transfer-learning, or gradient and model compressions. NMT models are resource-intensive both when training and deploying. We have examined methods of introducing approximations into NMT training to achieve improved linear scale-up between the computational resource (i.e., GPU) and the training speed, thereby reducing the training cost. We also have explored a method of approximating the NMT model to reduce its size.

Chapter 3 investigated the poor performance of the current state-of-the-art NMT architecture, the Transformer (Vaswani et al., 2017), when trained asynchronously. This chapter showed that stale updates and small mini-batch size damage the training convergence, especially of the former. We also found that RNN models were generally more stable toward such noise. We suggested accumulating the gradient updates sent by asynchronous workers in the server before applying parameter updates to mimic the behaviour of synchronous training while retaining asynchronous speed. Using this approach, we fixed the quality degradation in asynchronous Transformer training.

Chapter 4 and Chapter 5 addressed reducing the inter-worker network communication cost for parallel training. Each worker in parallel training communicates gradients to each other. In Chapter 4, we introduced a gradient compression algorithm by only exchanging the top 1% of the largest gradients (in absolute value), thereby significantly reducing the network communication cost. We followed an error feedback mechanism (Seide et al., 2014) where we store unsent gradients to be considered for the next iteration. We managed to train an RNN-based model using this approach with-

out sacrificing quality. Unfortunately, this approach introduced a noisy update. Thus, the model converged slower (reached the same quality measurement with more steps/training data). Moreover, we were unable to train a Transformer-based model under this gradient compression scheme since the error feedback mechanism introduced stale updates, which were shown to be harmful to the Transformer in Chapter 3.

Chapter 5 refined the gradient compression algorithm from Chapter 4 by incorporating local gradients. We added the locally-computed uncompressed gradient to the sparse gradient before updating. Then, we restored the gradient quality, which exhibited a better convergence rate compared to the vanilla gradient compression. Moreover, we managed to train a Transformer-based model using this approach. We further found that scaling the number of workers is challenging. First, we could not scale the learning rate and warm-up linear to the number of workers without sacrificing quality. Second, with more workers, the summed sparse gradients will become denser, thus reducing the compression efficiency. We found that the former issue was more prominent since we could not properly scale the baseline, which resulted in sub-linear speed improvement. However, we still gained a significant speed increase by introducing the gradient compression over that baseline.

We shifted our focus in Chapter 6 to investigate the use of cross-lingual transfer learning as a better model initialisation. We found that low-resource languages performed better even by transferring from unrelated language pairs, including a randomly generated language. However, the latter did not improve the quality as much. Transfer learning can be applied to a high-resource language as a better initialisation. We managed to train the model faster, though without any quality improvement.

In Chapter 7, we compressed the model with a 4-bit logarithmic quantisation to reduce the model size with an insignificant sacrifice in performance. The quantised model must be fine-tuned from a full precision model. Models trained in 4-bit precision de novo or those only quantising with a full-precision model performed poorly. We further explored 4-bit matrix multiplications for faster CPU inference. Quantising matrix multiplication operations slightly damaged the quality. Unfortunately, 4-bit log-based matrix multiplication is only as fast as 8-bit integer matrix multiplication since the current hardware did not support native 4-bit operations.

Throughout this thesis, we empirically showed that Transformer models are susceptible to noisy training conditions. In Chapter 3, we showed that a Transformer model could not be trained with an asynchronous SGD. However, reducing stale updates and increasing the batch size solved this issue. Chapter 5 also demonstrated

that a Transformer model could not be trained with sparse gradient updates. We then incorporated local gradients to re-construct dense gradients to resolve this issue. In Chapter 5 and 6, we showed that a Transformer model cannot be trained with aggressive learning rate or warm-up, which can be mitigated if the Transformer model is fine-tuned with transfer learning. Lastly, Chapter 7 showed that noise in the form of quantisation error degrades Transformer-based models more.

The error feedback mechanism (Seide et al., 2014), where we store quantisation error and re-add it back for the next iteration, was essential for all of our noisy training experiments. Without the error feedback mechanism, the model trained with sparse gradients (Chapter 3) diverged. However, the model could be trained without the error feedback if we incorporate local gradients from Chapter 5, though it yielded lower translation quality. Similarly, in Chapter 7, we showed that the error feedback mechanism must be applied when re-training the model under 4-bit precision to minimise quality degradation.

8.2 Future Work

Several ideas for future work related to this thesis include:

- **Exploring compression on different models and training configurations**

  We found that RNNs are more robust than Transformers to noisy gradients in multi-node training and quantization. An interesting future direction could be to widen this to more models, and analyse why some models are more robust to noise. Future studies could also experiment with different model sizes, such as larger Transformers (Huang et al., 2019), which have been shown to perform better while being more resource-demanding. In contrast, we can also explore stacking our compression techniques to smaller models or other compression technique (for example parameter sharing (Kim et al., 2019a), head prunning (Voita et al., 2019)) to achieve even better efficiency. This could lead to the development of models that are especially robust yet efficient, which would facilitate multi-device training and deployment on mobile devices.

  Future works could also extend these experiments to different hyperparameter configurations. While we used the Adam optimiser in all of our experiments, it might be interesting to investigate the interaction of different optimisers. Another hyperparameter that we are interested in exploring is the drop-out ratio.

Drop-out sets random activations to zero as noise to avoid overfitting. However, our compression techniques already introduce noise; therefore, this would be worth investigating.

- **Exploring Transfer Learning**

  In Chapter 6, we explored transfer learning as a better initialisation. Future work could generalise this approach by investigating improved Transformer initialisation without the need for training a parent model (e.g., with a different randomisation function). We also demonstrated that better initialisation enabled the Transformer to be trained with a more aggressive learning-rate warm-up. In contrast, in Chapter 5, we experienced difficulty in scaling up the learning rate and its warm-up in highly parallel training. With improved initialisation, it may be possible to better scale the hyperparameters to achieve faster training speed.

- **Layer-aware compression**

  Recent works have shown that each part of the parameter in a trained neural network is not equally important. For example, Voita et al. (2019) showed that some of the attention heads in Transformer can be pruned without affecting performance. Similarly, Kim et al. (2019a) mentioned that encoder layers are more sensitive to pruning compared to decoder layers. We can potentially connect this with our transfer learning results to determine whether the embedding layer can be ignored in transfer learning. We can apply these findings to our work to achieve improved gradient or model compression (Chapter 4 and 5) by compressing the least important layers more aggressively while preserving more important ones. We would also like to determine whether we can apply different compression rates for individual layers or components instead of only the global threshold. On a similar note, we quantised the model into 4-bit precision in Chapter 7, though less precision degraded the quality. However, it might be possible to use different precisions on different layers or components. While this idea has been partially explored since we did not quantise the biases at all, further exploration of this point could yield better model compression.

Bibliography

Alham Fikri Aji, Nikolay Bogoychev, Kenneth Heafield, and Rico Sennrich. 2020. In neural machine translation, what does transfer learning transfer? *ACL*.

Alham Fikri Aji and Kenneth Heafield. 2017. Sparse communication for distributed gradient descent. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, pages 440–445.

Alham Fikri Aji and Kenneth Heafield. 2019a. Making asynchronous stochastic gradient descent work for transformers. *EMNLP-IJCNLP 2019*, page 80.

Alham Fikri Aji and Kenneth Heafield. 2019b. Neural machine translation with 4-bit precision and beyond. *arXiv preprint arXiv:1909.06091*.

Alham Fikri Aji, Kenneth Heafield, and Nikolay Bogoychev. 2019. Combining global sparse gradients with local gradients in distributed neural network training. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 3617–3622.

Tolu Alabi, Jeffrey D Blanchard, Bradley Gordon, and Russel Steinbach. 2012. Fast k-selection algorithms for graphics processing units. *Journal of Experimental Algorithmics (JEA)*, 17:4–2.

Dan Alistarh, Demjan Grubic, Jerry Li, Ryota Tomioka, and Milan Vojnovic. 2017. Qsgd: Communication-efficient sgd via gradient quantization and encoding. In *Advances in Neural Information Processing Systems*, pages 1709–1720.

Dan Alistarh, Jerry Li, Ryota Tomioka, and Milan Vojnovic. 2016. Qsgd: Randomized quantization for communication-optimal stochastic gradient descent. *arXiv preprint arXiv:1610.02132*.

Sajid Anwar, Kyuyeon Hwang, and Wonyong Sung. 2015. Fixed point optimization of deep convolutional neural networks for object recognition. In *2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 1131–1135. IEEE.

Mikel Artetxe, Gorka Labaka, and Eneko Agirre. 2018. Unsupervised statistical machine translation. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, Brussels, Belgium. Association for Computational Linguistics.

Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. 2014. Neural machine translation by jointly learning to align and translate. In *ICLR 2015*.

Yoshua Bengio, Nicholas Léonard, and Aaron Courville. 2013. Estimating or propagating gradients through stochastic neurons for conditional computation. *arXiv preprint arXiv:1308.3432*.

Nikolay Bogoychev, Kenneth Heafield, Alham Fikri Aji, and Marcin Junczys-Dowmunt. 2018. Accelerating asynchronous stochastic gradient descent for neural machine translation. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 2991–2996.

Ondřej Bojar, Rajen Chatterjee, Christian Federmann, Yvette Graham, Barry Haddow, Shujian Huang, Matthias Huck, Philipp Koehn, Qun Liu, Varvara Logacheva, Christof Monz, Matteo Negri, Matt Post, Raphael Rubino, Lucia Specia, and Marco Turchi. 2017. Findings of the 2017 conference on machine translation (WMT17). In *Proceedings of the Second Conference on Machine Translation*, pages 169–214, Copenhagen, Denmark. Association for Computational Linguistics.

Ondřej Bojar, Christian Federmann, Mark Fishel, Yvette Graham, Barry Haddow, Matthias Huck, Philipp Koehn, and Christof Monz. 2018. Findings of the 2018 conference on machine translation (wmt18). In *Proceedings of the Third Conference on Machine Translation (WMT), Volume 2: Shared Task Papers*, pages 272–307. Association for Computational Linguistics.

Denny Britz, Anna Goldie, Minh-Thang Luong, and Quoc Le. 2017. Massive exploration of neural machine translation architectures. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, pages 1442–1451.

Jianmin Chen, Xinghao Pan, Rajat Monga, Samy Bengio, and Rafal Jozefowicz. 2016. Revisiting distributed synchronous sgd. *arXiv preprint arXiv:1604.00981*.

Mia Xu Chen, Orhan Firat, Ankur Bapna, Melvin Johnson, Wolfgang Macherey, George Foster, Llion Jones, Mike Schuster, Noam Shazeer, Niki Parmar, et al. 2018. The best of both worlds: Combining recent advances in neural machine translation. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 76–86.

Yong Cheng, Yang Liu, Qian Yang, Maosong Sun, and Wei Xu. 2016. Neural machine translation with pivot languages. *arXiv preprint arXiv:1611.04928*.

Trishul M Chilimbi, Yutaka Suzue, Johnson Apacible, and Karthik Kalyanaraman. 2014. Project adam: Building an efficient and scalable deep learning training system. In *OSDI*, volume 14, pages 571–582.

Kyunghyun Cho, Bart van Merrienboer, Çaglar Gülçehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. 2014. Learning phrase representations using RNN encoder-decoder for statistical machine translation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing, EMNLP 2014, October 25-29, 2014, Doha, Qatar, A meeting of SIGDAT, a Special Interest Group of the ACL*, pages 1724–1734. ACL.

Anna Currey, Antonio Valerio Miceli Barone, and Kenneth Heafield. 2017. Copied monolingual data improves low-resource neural machine translation. In *Proceedings of the Second Conference on Machine Translation*, pages 148–156.

Raj Dabre, Tetsuji Nakagawa, and Hideto Kazawa. 2017. An empirical study of language relatedness for transfer learning in neural machine translation. *on Language, Information and Computation*, page 282.

Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Andrew Senior, Paul Tucker, Ke Yang, Quoc V Le, et al. 2012. Large scale distributed deep networks. In *Advances in neural information processing systems*, pages 1223–1231.

Chenchen Ding, Hnin Thu Zar Aye, Win Pa Pa, Khin Thandar Nwet, Khin Mar Soe, Masao Utiyama, and Eiichiro Sumita. 2019. Towards Burmese (Myanmar) morphological analysis: Syllable-based tokenization and part-of-speech tagging. *ACM*

Transactions on Asian and Low-Resource Language Information Processing (TALLIP), 19(1):5.

Chenchen Ding, Masao Utiyama, and Eiichiro Sumita. 2018. NOVA: A feasible and flexible annotation system for joint tokenization and part-of-speech tagging. *ACM Transactions on Asian and Low-Resource Language Information Processing (TALLIP)*, 18(2):17.

Nikoli Dryden, Sam Ade Jacobs, Tim Moon, and Brian Van Essen. 2016. Communication quantization for data-parallel training of deep neural networks. In *Proceedings of the Workshop on Machine Learning in High Performance Computing Environments*, pages 1–8. IEEE Press.

Sanghamitra Dutta, Gauri Joshi, Soumyadip Ghosh, Parijat Dube, and Priya Nagpurkar. 2018. Slow and stale gradients can win the race: Error-runtime trade-offs in distributed sgd. *arXiv preprint arXiv:1803.01113*.

Philip Gage. 1994. A new algorithm for data compression. *The C Users Journal*, 12(2):23–38.

Mozhdeh Gheini and Jonathan May. 2019. A universal parent model for low-resource neural machine translation transfer. *arXiv preprint arXiv:1909.06516*.

Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. 2017. Accurate, large minibatch sgd: Training imagenet in 1 hour. *arXiv preprint arXiv:1706.02677*.

Jiatao Gu, Hany Hassan, Jacob Devlin, and Victor O.K. Li. 2018a. Universal neural machine translation for extremely low resource languages. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers)*, pages 344–354, New Orleans, Louisiana. Association for Computational Linguistics.

Jiatao Gu, Yong Wang, Yun Chen, Victor O. K. Li, and Kyunghyun Cho. 2018b. Meta-learning for low-resource neural machine translation. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 3622–3631, Brussels, Belgium. Association for Computational Linguistics.

Suyog Gupta, Wei Zhang, and Fei Wang. 2016. Model accuracy and runtime tradeoff in distributed deep learning: A systematic study. In *2016 IEEE 16th International Conference on Data Mining (ICDM)*, pages 171–180. IEEE.

Song Han, Huizi Mao, and William J Dally. 2015. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149*.

Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation*, 9(8):1735–1780.

Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, et al. 2019. Gpipe: Efficient training of giant neural networks using pipeline parallelism. In *Advances in Neural Information Processing Systems*, pages 103–112.

Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. 2016. Binarized neural networks. In *Advances in neural information processing systems*, pages 4107–4115.

Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. 2017. Quantized neural networks: Training neural networks with low precision weights and activations. *The Journal of Machine Learning Research*, 18(1):6869–6898.

Kyuyeon Hwang and Wonyong Sung. 2014. Fixed-point feedforward deep neural network design using weights+ 1, 0, and- 1. In *2014 IEEE Workshop on Signal Processing Systems (SiPS)*, pages 1–6. IEEE.

Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. 2018. Quantization and training of neural networks for efficient integer-arithmetic-only inference. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2704–2713.

Melvin Johnson, Mike Schuster, Quoc V. Le, Maxim Krikun, Yonghui Wu, Zhifeng Chen, Nikhil Thorat, Fernanda B. Viégas, Martin Wattenberg, Greg Corrado, Macduff Hughes, and Jeffrey Dean. 2016. Google's multilingual neural machine translation system: Enabling zero-shot translation. *CoRR*.

Marcin Junczys-Dowmunt. 2019. Microsoft translator at wmt 2019: Towards large-scale document-level neural machine translation. In *Proceedings of the Fourth Conference on Machine Translation (Volume 2: Shared Task Papers, Day 1)*, pages 225–233.

Marcin Junczys-Dowmunt, Tomasz Dwojak, and Hieu Hoang. 2016. Is neural machine translation ready for deployment? A case study on 30 translation directions. In *Program of the 13th International Workshop on Spoken Language Translation (IWSLT 2016)*.

Marcin Junczys-Dowmunt, Kenneth Heafield, Hieu Hoang, Roman Grundkiewicz, and Anthony Aue. 2018. Marian: Cost-effective high-quality neural machine translation in c++. In *Proceedings of the 2nd Workshop on Neural Machine Translation and Generation*, pages 129–135.

Young Jin Kim, Marcin Junczys-Dowmunt, Hany Hassan, Alham Fikri Aji, Kenneth Heafield, Roman Grundkiewicz, and Nikolay Bogoychev. 2019a. From research to production and back: Ludicrously fast neural machine translation. In *Proceedings of the 3rd Workshop on Neural Generation and Translation*, pages 280–288.

Yunsu Kim, Yingbo Gao, and Hermann Ney. 2019b. Effective cross-lingual transfer of neural machine translation models without shared vocabularies. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 1246–1257, Florence, Italy. Association for Computational Linguistics.

Yunsu Kim, Jiahui Geng, and Hermann Ney. 2018. Improving unsupervised word-by-word translation with language model and denoising autoencoder. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 862–868, Brussels, Belgium. Association for Computational Linguistics.

Diederik P Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

Tom Kocmi and Ondřej Bojar. 2018. Trivial transfer learning for low-resource neural machine translation. In *Proceedings of the Third Conference on Machine Translation: Research Papers*, pages 244–252, Belgium, Brussels. Association for Computational Linguistics.

Jakub Konečnỳ, H Brendan McMahan, Felix X Yu, Peter Richtárik, Ananda Theertha Suresh, and Dave Bacon. 2016. Federated learning: Strategies for improving communication efficiency. *arXiv preprint arXiv:1610.05492*.

Guillaume Lample, Myle Ott, Alexis Conneau, Ludovic Denoyer, and Marc'Aurelio Ranzato. 2018. Phrase-based & neural unsupervised machine translation. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 5039–5049, Brussels, Belgium. Association for Computational Linguistics.

Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. 1998. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324.

Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. 2016. Layer normalization. *arXiv preprint arXiv:1607.06450*.

Xiangru Lian, Yijun Huang, Yuncheng Li, and Ji Liu. 2015. Asynchronous parallel stochastic gradient for nonconvex optimization. In *Advances in Neural Information Processing Systems*, pages 2737–2745.

Darryl Lin, Sachin Talathi, and Sreekanth Annapureddy. 2016. Fixed point quantization of deep convolutional networks. In *International Conference on Machine Learning*, pages 2849–2858.

Yu-Hsiang Lin, Chian-Yu Chen, Jean Lee, Zirui Li, Yuyan Zhang, Mengzhou Xia, Shruti Rijhwani, Junxian He, Zhisong Zhang, Xuezhe Ma, Antonios Anastasopoulos, Patrick Littell, and Graham Neubig. 2019. Choosing transfer languages for cross-lingual learning. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 3125–3135, Florence, Italy. Association for Computational Linguistics.

Yujun Lin, Song Han, Huizi Mao, Yu Wang, and Bill Dally. 2018. Deep gradient compression: Reducing the communication bandwidth for distributed training. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*.

Brendan McMahan, Eider Moore, Daniel Ramage, Seth Hampson, and Blaise Aguera y Arcas. 2017. Communication-efficient learning of deep networks from decentralized data. In *Artificial Intelligence and Statistics*, pages 1273–1282.

Brendan McMahan and Matthew Streeter. 2014. Delay-tolerant algorithms for asynchronous distributed online learning. In *Advances in Neural Information Processing Systems*, pages 2915–2923.

Antonio Valerio Miceli-Barone, Jindřich Helcl, Rico Sennrich, Barry Haddow, and Alexandra Birch. 2017. Deep architectures for neural machine translation. In *Proceedings of the Second Conference on Machine Translation*, pages 99–107.

Daisuke Miyashita, Edward H Lee, and Boris Murmann. 2016. Convolutional neural networks using logarithmic data representation. *arXiv preprint arXiv:1603.01025*.

Ramón P Ñeco and Mikel L Forcada. 1996. Beyond mealy machines: Learning translators with recurrent neural networks. In *Proceedings of the 1996 International Neural Network Society Annual Meeting*, San Diego, California, USA.

Toan Q Nguyen and David Chiang. 2017. Transfer learning across low-resource, related languages for neural machine translation. In *Proceedings of the Eighth International Joint Conference on Natural Language Processing (Volume 2: Short Papers)*, pages 296–301.

Toan Q Nguyen and Julian Salazar. 2019. Transformers without tears: Improving the normalization of self-attention. *arXiv preprint arXiv:1910.05895*.

Myle Ott, Sergey Edunov, David Grangier, and Michael Auli. 2018. Scaling neural machine translation. In *Proceedings of the Third Conference on Machine Translation: Research Papers*, pages 1–9.

Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002a. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting on association for computational linguistics*, pages 311–318. Association for Computational Linguistics.

Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002b. BLEU: A method for automatic evalution of machine translation. In *Proceedings 40th Annual Meeting of the Association for Computational Linguistics*, pages 311–318, Philadelphia, PA.

Emmanouil Antonios Platanios, Otilia Stretcu, Graham Neubig, Barnabas Poczos, and Tom Mitchell. 2019. Competence-based curriculum learning for neural machine

translation. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 1162–1172.

Martin Popel and Ondřej Bojar. 2018. Training tips for the transformer model. *The Prague Bulletin of Mathematical Linguistics*, 110(1):43–70.

Matt Post. 2018. A call for clarity in reporting bleu scores. In *Proceedings of the Third Conference on Machine Translation: Research Papers*, pages 186–191.

Ye Qi, Devendra Singh Sachan, Matthieu Felix, Sarguna Janani Padmanabhan, and Graham Neubig. 2018. When and why are pre-trained word embeddings useful for neural machine translation? *arXiv preprint arXiv:1804.06323*.

Jerry Quinn and Miguel Ballesteros. 2018. Pieces of eight: 8-bit neural machine translation. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 3 (Industry Papers)*, pages 114–120.

Rajat Raina, Anand Madhavan, and Andrew Y Ng. 2009. Large-scale deep unsupervised learning using graphics processors. In *Proceedings of the 26th annual international conference on machine learning*, pages 873–880. ACM.

Benjamin Recht, Christopher Re, Stephen Wright, and Feng Niu. 2011. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *Advances in neural information processing systems*, pages 693–701.

Abigail See, Minh-Thang Luong, and Christopher D Manning. 2016. Compression of neural machine translation models via pruning. *arXiv preprint arXiv:1606.09274*.

Frank Seide, Hao Fu, Jasha Droppo, Gang Li, and Dong Yu. 2014. 1-bit stochastic gradient descent and application to data-parallel distributed training of speech DNNs. In *Interspeech*.

Rico Sennrich, Alexandra Birch, Anna Currey, Ulrich Germann, Barry Haddow, Kenneth Heafield, Antonio Valerio Miceli Barone, and Philip Williams. 2017. The University of Edinburgh's neural mt systems for WMT17. In *Proceedings of the Second Conference on Machine Translation*, pages 389–399.

Rico Sennrich, Barry Haddow, and Alexandra Birch. 2016a. Edinburgh neural machine translation systems for WMT 16. In *Proceedings of the ACL 2016 First Conference on Machine Translation (WMT16)*.

Rico Sennrich, Barry Haddow, and Alexandra Birch. 2016b. Improving neural machine translation models with monolingual data. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 86–96, Berlin, Germany. Association for Computational Linguistics.

Rico Sennrich, Barry Haddow, and Alexandra Birch. 2016c. Neural machine translation of rare words with subword units. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics*, pages 1715–1725.

Samuel L Smith, Pieter-Jan Kindermans, Chris Ying, and Quoc V Le. 2017. Don't decay the learning rate, increase the batch size. *arXiv preprint arXiv:1711.00489*.

Samuel L Smith and Quoc V Le. 2017. A bayesian perspective on generalization and stochastic gradient descent. *arXiv preprint arXiv:1710.06451*.

Anders Søgaard, Sebastian Ruder, and Ivan Vulić. 2018. On the limitations of unsupervised bilingual dictionary induction. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 778–788, Melbourne, Australia. Association for Computational Linguistics.

Anand Srinivasan, Ajay Jain, and Parnian Barekatain. 2018. An analysis of the delayed gradients problem in asynchronous SGD.

Nikko Strom. 2015. Scalable distributed dnn training using commodity gpu cloud computing. In *INTERSPEECH*, volume 7, page 10.

Ilya Sutskever, Oriol Vinyals, and Quoc V Le. 2014. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*, pages 3104–3112.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in Neural Information Processing Systems*, pages 5998–6008.

Elena Voita, David Talbot, Fedor Moiseev, Rico Sennrich, and Ivan Titov. 2019. Analyzing multi-head self-attention: Specialized heads do the heavy lifting, the rest can be pruned. *arXiv preprint arXiv:1905.09418*.

Chong Wang, Xi Chen, Alexander J Smola, and Eric P Xing. 2013. Variance reduction for stochastic gradient optimization. In *Advances in Neural Information Processing Systems*, pages 181–189.

Wei Wen, Cong Xu, Feng Yan, Chunpeng Wu, Yandan Wang, Yiran Chen, and Hai Li. 2017. Terngrad: Ternary gradients to reduce communication in distributed deep learning. In *Advances in neural information processing systems*, pages 1509–1519.

Wei Zhang, Suyog Gupta, Xiangru Lian, and Ji Liu. 2016. Staleness-aware async-sgd for distributed deep learning. In *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence*, pages 2350–2356. AAAI Press.

Martin Zinkevich, Markus Weimer, Lihong Li, and Alex J Smola. 2010. Parallelized stochastic gradient descent. In *Advances in neural information processing systems*, pages 2595–2603.

Barret Zoph, Deniz Yuret, Jonathan May, and Kevin Knight. 2016. Transfer learning for low-resource neural machine translation. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, pages 1568–1575, Austin, Texas. Association for Computational Linguistics.

Daniel Zwillinger and Stephen Kokoska. 1999. *CRC standard probability and statistics tables and formulae*. CRC Press.